Broadview®
www.broadview.com.cn

Spring社区倾力打造的微服务架构开源解决方案
助力快速构建微服务架构体系

Spring Cloud
微服务实战

翟永超 著

中国工信出版集团　電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VISIT…

LANZAROTE
Caliente.COM

Spring Cloud
微服务实战

翟永超 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书从时下流行的微服务架构概念出发，详细介绍了Spring Cloud针对微服务架构中几大核心要素的解决方案和基础组件。对于各个组件的介绍，本书主要以示例与源码结合的方式来帮助读者更好地理解这些组件的使用方法以及运行原理。同时，在介绍的过程中，还包含了作者在实践中所遇到的一些问题和解决思路，可供读者在实践中作为参考。

本书适合所有Java开发人员，尤其适合正在做微服务架构技术选型或正在实施微服务架构的团队查阅和参考。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Spring Cloud微服务实战 / 翟永超著. —北京：电子工业出版社，2017.5
ISBN 978-7-121-31301-1

Ⅰ. ①S… Ⅱ. ①翟… Ⅲ. ①互联网络－网络服务器 Ⅳ. ①TP368.5

中国版本图书馆CIP数据核字（2017）第071305号

策划编辑：张春雨
责任编辑：刘　舫
印　　刷：北京天宇星印刷厂
装　　订：北京天宇星印刷厂
出版发行：电子工业出版社
北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　印张：27.5　字数：586千字
版　　次：2017年5月第1版
印　　次：2017年5月第1次印刷
定　　价：89.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

推荐序一

我和 Java 很有缘，2003 年在富士通南大实习的时候，就开始用 Struts/JSP/Hibernate/MySQL 做第一个 Java 项目——SPIF（http://www.fujitsu.com/cn/products/software/applications/spif/）。

工作之后，主要做 J2EE 的开发，并开始慢慢接触和使用 Spring。加入 EMC 之后，很有幸和 Spring 成为一个大家庭（EMC 收购了 VMware，VMware 收购了 Spring）。2012 年，我和 Spring Data/XD 的负责人 Mark Pollack 在中国开过一次小会，和他探讨了一些关于 Data Pipeline 的想法。后来我也看了很多 Josh Long（龙应春）的演讲视频，并在 2016 年终于有幸能够与他在一个技术大会上同场演讲，并在会议之后做了很多深入沟通。

2015 年上半年，我读了两本书：*The Phoenix Project* 和 *Migrating to Cloud-Native Application Architectures*，让我对 DevOps、微服务和云原生架构有了初步的认识，也让我对 Netflix 的那套 OSS 套件有了一个初步印象。

我是在 2015 年 9 月加入麻袋理财之后开始接触 Spring Boot 的，试用之后感觉它很神奇，再也没有被 Spring 之前那些烦琐配置所束缚。当时正好和一个架构师讨论要做一个项目的升级改造，决定采用 Spring Boot 和微服务架构。开始的时候，服务治理还是用了 Dubbo。之后因为对 Spring Cloud 有了比较深刻的认识，在之后一个全新项目上，我们完全按照微服务架构，使用 Spring Boot 和 Cloud 进行开发，并采用 CI / CD 自动化流程和容器化部署。

因为使用了 Spring Cloud，让我对 Spring Cloud 的相关信息特别关注。一个偶然的机会，我认识了 Spring Cloud 中国社区的负责人许进、翟永超（本书作者）和周立，探讨了很多使用 Spring Cloud 的经验，感觉与他们和 Spring Cloud 相见恨晚。

翟永超本人写了很多关于 Spring Cloud 使用的博客，不同于一般作者，他写的内容更

加贴近实际，是自己工作经验的深刻总结，可以拿来直接用于生产。

有一次我们聊到关于配置中心（Spring Cloud Config）如何在生产中使用，他解答了我很多问题，并告诉我他写了一本书，书中就会包含这些内容。这让我对这本书充满期待。后面也有幸见到了本人，一个瘦瘦高高的书生，一看就是一个很有内涵的技术人。畅聊之后，翟永超就把书发给了我，让我先睹为快。

我把翟永超的书仔细拜读了一遍，最大的收获就是让我对 Spring Cloud 的认识又上升了一个层次。我之前对 Spring Cloud 的理解更多的是知其然，但是却不知道其所以然，对 Spring Cloud 里面的逻辑知之甚少。而读了翟永超的《Spring Cloud 微服务实战》一书后，让我对 Spring Cloud 各个组件的认识提升了一个层次，同时也让我对 Spring Cloud 各个组件的实现原理有了初步的认识，因此我建议所有打算将 Spring Cloud 用于生产的朋友一定要好好读一下这本书。

王天青

DaoCloud 首席架构师

2017 年 3 月

推荐序二

2016 年 10 月开始，我在冰鉴科技负责微服务架构迁移相关的调研和筹建工作。我比较了 Dubbo、Dubbox、Motan、Spring Cloud 等框架后，最终锁定在 Spring Cloud 上。这是一个非常年轻的框架，关于它的中文文档少之又少，更不用说有深度的技术干货了。

当我的团队在利用搜索引擎进行相关检索时，永超的技术博客十分显眼地排在了前列，我非常感激他贡献的这个系列的文章，这在我们团队做微服务架构迁移的工作中，起到了关键作用。后来有一天，我俩在一个架构群中相识，一番讨论后发现是博主本人并且他有写书计划时，我告诉了我的团队，我们不谋而合地决定要在该书出版时迅速收入囊中，做到人手一本。而今天对于我来说更是非常荣幸，能够给永超的新书写推荐序。

Spring Cloud 是一个微服务架构实施的综合性解决框架，而在如何构建微服务的选择上，由于我们团队是从 SSM（Spring + Spring MVC + MyBatis）框架开始演进的，基于让演进中改动最小的初衷，我们决定使用 Spring Boot 做微服务构建。我们从对 Spring Boot 的调研开始就一直关注着永超的技术博客，在第一次接触 Spring Boot 的时候就被它“习惯优于配置”的设计概念深深吸引，这无疑简化了做业务逻辑开发同事的工作量，也使得他们可以不用关注配置细节。本书中也有关于 Spring Boot 基础知识的详细讲解以及一个案例工程带你快速构建属于你的第一个微服务。

如开头所述，为了将系统微服务化，我们也一直在对 Spring Cloud 进行相关调研。这本书也是国内市场上为数不多的、全面讲解 Spring Cloud 微服务的中文图书。本书详细讲解了 Spring Cloud 生态的各类组件，涵盖了服务治理组件 Eureka、客户端负载均衡组件 Ribbon、服务容错保护组件 Hystrix、声明式服务调用组件 Feign、API 网关治理组件 Zuul、分布式配置中心组件 Config、消息总线组件 Bus、消息驱动组件 Stream、分布式服务跟踪组件 Sleuth。这包含了我们在实施微服务中需要深入了解的各个轮子，是一本需要仔细

研读、反复阅读的精品之作。

最后，预祝永超在 Spring Cloud 的学习和工作中再创佳绩，也希望读者朋友能够在阅读完本书后快速地搭建好实施微服务过程中的基础脚手架，并在未来工作中能够将团队的一些实践通过 Spring Cloud 中国社区进行交流，为开源贡献自己的一份力量。

朱清

冰鉴科技信息技术部总监

Spring Cloud 中国社区联合创始人

2017.03.27

推荐序三

收到本书作者翟永超的邀请为这本书写推荐序，其实我是很谨慎的。抱着对读者负责、对技术严谨的态度，不能在完全不懂 Spring Cloud 的基础上妄加评论。就像 2009 年的云计算和现在的大数据，“Big data is like teenage sex: everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it.”。所以我概读了书中的内容，的确是一本好书，特别是在基于技术实践的阐述中又不失对“微服务化”理论层面的讲解以及发展演进过程的说明。

结合在云计算行业中为大量企业级客户做的服务案例，“集中化”的系统架构确实在企业级客户业务中受到越来越多的挑战，随着业务变化对 IT 需求的不断增加，处于逐渐失控的状态。CIO 们受到越来越大的挑战，希望做到数据驱动业务，那第一个阶段就要做去中心化的改造。如书中所阐述，“微服务化”其实并不是简单的技术革新，而是对团队组织，系统架构，系统研发，自动化测试、发布、运维都提出了一系列的变革要求。所以我觉得，不管是架构师、运维经理、研发主管还是 CIO 都可以从本书中有所收获。

同样，阿里云的企业级中间件 EDAS（基于阿里系的 Dubbo 开源项目）配合强大的飞天云平台与 Docker 服务的支持，在大中型企业客户业务中得到更多的验证，如森马服饰、来伊份、正佳广场、中石化的易派客电商平台等。与这些商业化的中间件产品相比，Spring Cloud 得到了更多热衷开源项目的人的支持，相信在有足够团队技术能力的保障下，也会取得越来越多的成功案例。书如其人，值得认真拜读，我会推荐给更多的人，为翟永超点赞。

李俊涛

上海驻云科技执行总监

推荐语

本书从时下流行的微服务架构概念出发，结合 Spring Cloud 的解决方案，深入浅出地剖析了其在构建微服务架构中所需的各个基础设施和技术要点，包括服务治理、容错保护、API 网关、配置管理、消息总线等。作者不仅对如何使用各个组件做了详细介绍，还从原理上做了很多分析，可以帮助读者更好地理解 Spring Cloud 的运行原理，这有助于我们在实战中有效地排错和做进一步扩展。本书是微服务架构方面非常不错的实战书籍，强烈推荐正在做微服务实践或打算实施微服务的团队作为参考资料。

南志文

百联全渠道研发总监

Spring Cloud 的诞生对于没有足够资金投入或者技术储备实力的技术团队是一种福音。利用 Spring Cloud 的一站式解决方案，可以很轻松地搭建起微服务架构的软件系统，大大减少了开发成本，从容应对业务的快速发展。本书是国内第一本 Spring Cloud 的实战书籍，给我带来了无限惊喜。作者由浅入深地讲解了基于 Spring Cloud 构建微服务所需要的各个核心组件，并配有大量实战代码，理论和实践兼备，读后收获颇丰。强烈推荐给每一位对 Spring Cloud 感兴趣或是打算使用 Spring Cloud 的技术人员阅读。

覃罗春

德比软件产品开发负责人

当下最火的词无外乎就是“微服务”了，但是很多创业公司想要实现微服务架构体系还需要做很多方面的工作才可以逐步实现，所需花费的成本还是较大的。而近年来 Spring

Boot/Cloud 生态架构体系的出现为行业提供了一站式解决方案，解决了不少公司的架构选型和维护方面的难题。本书是国内第一本以Spring Cloud为技术蓝本的微服务类实战书籍，不仅结合实际案例介绍了 Spring Cloud 的使用，还从源码的角度深入分析了原理实现，强烈推荐每一位开发者和架构师收藏和学习。

程超

合众支付资深技术专家

随着微服务架构的兴起，企业 IT 架构开始变革，国内出现首批微服务实战布道者。本书作者翟永超，作为 Spring Cloud 中国社区联合发起人和国内首批 Spring Cloud 实践与布道者，发表的博文超过数百万次访问量。他的《Spring Cloud 微服务实战》一书，包含大量生产实战经验，把 Spring Cloud 常用组件通过案例剖析，可帮助企业和开发者快速实施微服务架构。

许进（xujin.org）

Spring Cloud 中国社区创始人

中间件高级研发工程师

近几年，微服务概念逐渐深入人心，国内各家互联网公司都引入了相应的实践。而被应用最多的就是 Spring Cloud 这套被戏称为“全家桶”的微服务框架。它几乎实现了微服务的所有功能，而且又完美符合微服务的基础理论，可帮助大家提高工作效率。但是，国内关于 Spring Cloud 的中文资料相对比较匮乏，很多学习者遍寻入门而不得。在此大背景下，有一些有识之士无私贡献了自己的绵薄之力，本书作者翟永超就是其中一位。这本书对于广大需要在公司中实践微服务的人们来说绝对是一本可以快速上手实现微服务的工作手册。我希望这本书犹如一颗种子，能在国内互联网环境的土壤中生根发芽，最后变成一棵参天大树。

吴峻申

上海青客机器人有限公司架构师

前言

“微服务”架构在这几年被广泛传播，变得非常火热，以至于关于微服务架构相关的开源框架和工具都变得越来越活跃，比如：Netflix OSS、Dubbo、Apache Thrift 等。Spring Cloud 也因为 Spring 社区在企业应用领域的广泛知名度和强大影响力，受到了广大架构师与开发者的高度关注。

从接触 Spring Cloud 开始，我除了被其庞大的项目结构震撼之外，还被其所要完成的远大目标所吸引。该项目不同于其他 Spring 的优秀项目，它不再是一个基础框架类，而是一个更高层次的、架构视角的综合性大型项目，其目标旨在构建一套标准化的微服务解决方案，让架构师、开发者在使用微服务理念构建应用系统的时候，面对各个环节的问题都可以找到相应的组件来处理。引用网友戏称的一个比喻：Spring Cloud 可以说是 Spring 社区为微服务架构提供的一个“全家桶”套餐。由于“套餐”中的组件通过一个社区进行包装与整合，使得“套餐”中各个组件之间的配合变得更加和谐，这可以有效减少我们在组件的选型和整合上花费的精力，所以它可以帮助我们快速构建起基础的微服务架构系统。

虽然，Spring Cloud 提供了很多我们期待的内容，但是因其涵盖的内容非常广泛，并且知识跨度较大，因此对于很多初学者来说就像被专业名词轰炸了一样，入门的难度也就大大提高了。同时，中文文档与资料的匮乏，以及官方文档的内容对于使用描述并不够细致等问题，也直接提升了使用者的学习门槛。这些看似都不是什么大问题，但是却在一定程度上阻碍了 Spring Cloud 在国内的推广与发展，毕竟任何一项优秀技术都需要有大批的实践者才能得到不断优化、完善和发扬光大。作为一名 Spring 社区的忠实粉丝和长期实践者，自然希望可以有更多的开发者可以参与到 Spring Cloud 的使用和贡献中来，笔者也就萌生了想要编写一些入门文章的念头，一方面对自身知识的掌握做一些整理，另一方面也希望这些内容可以成为后来者的学习资料。于是就开始坚持着写了一些基础的入门文章和示例，没有想到会受到不少 Spring 爱好者的持续关注，在创建了相关的 QQ 交流群之后，短短一个月的时间，交流群的人数就突破了 1000 人。由于在交流过程中发现很多问题重复出现，

而这些问题并没有得到很好的整理，也没有办法被搜索引擎收录，于是就创建了 Spring Cloud 中文社区论坛，以帮助收集交流过程中提出和解决的各种问题，方便将来学习者可以搜索到这些前人踩过的坑。

之后，有幸在电子工业出版社计算机出版分社的张春雨先生的邀请下，开始编写这本关于 Spring Cloud 的入门书籍。在这本书的编写期间，由于工作、家庭等因素，使得与大家交流的时间变得越来越少，但好在有诸多网友和热心爱好者帮忙一起维护着交流群与论坛，为大家提供了很多宝贵的学习资源，我也从中得到了不少启发和收获。同时，感谢后来建议并牵头整合目前国内 Spring Cloud 学习资源的许进，他在此期间承担了很多沟通和网站维护工作，为 Spring Cloud 中国社区付出了不少精力，后续我也会重新加入进来，继续编写在线免费入门教程，以帮助更多的爱好者快速入门 Spring Cloud。我们也欢迎更多的爱好者参与进来，一起为社区贡献力量，以帮助 Spring Cloud 在国内被更好地应用与成长。

轻松注册成为博文视点社区用户（www.broadview.com.cn），即可享受以下服务：

下载资源：本书所提供的示例代码及资源文件均可在“下载资源”处下载。

提交勘误：您对书中内容的修改意见可在“提交勘误”处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

与作者交流：在页面下方“读者评论”处留下您的疑问或观点，与作者和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31301

二维码：

目录

第 1 章　基础知识 1
什么是微服务架构 1
与单体系统的区别 1
如何实施微服务 2
为什么选择 Spring Cloud 6
Spring Cloud 简介 7
版本说明 8

第 2 章　微服务构建：Spring Boot 11
框架简介 12
快速入门 13
项目构建与解析 13
实现 RESTful API 17
配置详解 20
配置文件 20
自定义参数 22
参数引用 22
使用随机数 23
命令行参数 23
多环境配置 24
加载顺序 25

监控与管理 26
初识 actuator 27
原生端点 28
小结 38

第 3 章 服务治理：Spring Cloud Eureka 39

服务治理 39
Netflix Eureka 40
搭建服务注册中心 41
注册服务提供者 43
高可用注册中心 46
服务发现与消费 48
Eureka 详解 51
基础架构 52
服务治理机制 52
源码分析 56
配置详解 65
服务注册类配置 65
服务实例类配置 67
跨平台支持 71

第 4 章 客户端负载均衡：Spring Cloud Ribbon 73

客户端负载均衡 73
RestTemplate 详解 75
GET 请求 75
POST 请求 77
PUT 请求 79
DELETE 请求 79
源码分析 80
负载均衡器 91
负载均衡策略 109
配置详解 123

自动化配置 124
Camden 版本对 RibbonClient 配置的优化 125
参数配置 127
与 Eureka 结合 127
重试机制 128

第 5 章 服务容错保护：Spring Cloud Hystrix 130

快速入门 131
原理分析 135
工作流程 135
断路器原理 144
依赖隔离 148
使用详解 151
创建请求命令 151
定义服务降级 154
异常处理 157
命令名称、分组以及线程池划分 158
请求缓存 159
请求合并 166
属性详解 172
Command 属性 174
collapser 属性 184
threadPool 属性 185
Hystrix 仪表盘 187
Turbine 集群监控 192
构建监控聚合服务 192
与消息代理结合 196

第 6 章 声明式服务调用：Spring Cloud Feign 199

快速入门 200
参数绑定 202
继承特性 205

Ribbon 配置 209
全局配置 209
指定服务配置 209
重试机制 210
Hystrix 配置 211
全局配置 211
禁用 Hystrix 211
指定命令配置 212
服务降级配置 212
其他配置 214
请求压缩 214
日志配置 215

第 7 章　API 网关服务：Spring Cloud Zuul 217

快速入门 219
构建网关 220
请求路由 221
请求过滤 223
路由详解 226
传统路由配置 226
服务路由配置 228
服务路由的默认规则 229
自定义路由映射规则 229
路径匹配 230
路由前缀 233
本地跳转 234
Cookie 与头信息 235
Hystrix 和 Ribbon 支持 236
过滤器详解 238
过滤器 238
请求生命周期 239
核心过滤器 240
异常处理 244

禁用过滤器 256
动态加载 257
动态路由 257
动态过滤器 261

第 8 章 分布式配置中心：Spring Cloud Config 267

快速入门 267
构建配置中心 268
配置规则详解 269
客户端配置映射 272
服务端详解 274
基础架构 274
Git 配置仓库 276
SVN 配置仓库 279
本地仓库 279
本地文件系统 279
健康监测 280
属性覆盖 281
安全保护 281
加密解密 282
高可用配置 286
客户端详解 286
URI 指定配置中心 287
服务化配置中心 287
失败快速响应与重试 290
获取远程配置 292
动态刷新配置 293

第 9 章 消息总线：Spring Cloud Bus 295

消息代理 295
RabbitMQ 实现消息总线 296
基本概念 297

安装与使用 298
快速入门 302
整合 Spring Cloud Bus 306
原理分析 307
指定刷新范围 308
架构优化 309
RabbitMQ 配置 310
Kafka 实现消息总线 312
Kafka 简介 312
快速入门 313
整合 Spring Cloud Bus 315
Kafka 配置 318
深入理解 318
源码分析 320
其他消息代理的支持 342

第 10 章 消息驱动的微服务：Spring Cloud Stream 344

快速入门 344
核心概念 349
绑定器 350
发布-订阅模式 351
消费组 353
消息分区 354
使用详解 355
开启绑定功能 355
绑定消息通道 356
消息生产与消费 360
响应式编程 366
消费组与消息分区 368
消息类型 370
绑定器详解 373
绑定器 SPI 373
自动化配置 374

多绑定器配置 374
RabbitMQ 与 Kafka 绑定器 376
配置详解 376
基础配置 377
绑定通道配置 377
绑定器配置 379

第 11 章 分布式服务跟踪：Spring Cloud Sleuth 386

快速入门 386
准备工作 386
实现跟踪 389
跟踪原理 390
抽样收集 392
与 Logstash 整合 394
与 Zipkin 整合 397
HTTP 收集 398
消息中间件收集 402
收集原理 404
数据存储 414
API 接口 417

附录 A Starter POMs 419

后记 421

第 1 章

基础知识

在进行 Spring Cloud 的具体内容介绍之前，我们先通过本章学习一些关于微服务架构以及 Spring Cloud 的基础知识。对 Spring Cloud 能够解决的具体问题有一个大致的了解，以帮助我们更好地理解后续章节对各个组件的介绍。

什么是微服务架构

“微服务”一词源于 Martin Fowler 的名为 *Microservices* 的博文，可以在他的官方博客上找到：`http://martinfowler.com/articles/microservices.html`。

简单地说，微服务是系统架构上的一种设计风格，它的主旨是将一个原本独立的系统拆分成多个小型服务，这些小型服务都在各自独立的进程中运行，服务之间通过基于 HTTP 的 RESTful API 进行通信协作。被拆分成的每一个小型服务都围绕着系统中的某一项或一些耦合度较高的业务功能进行构建，并且每个服务都维护着自身的数据存储、业务开发、自动化测试案例以及独立部署机制。由于有了轻量级的通信协作基础，所以这些微服务可以使用不同的语言来编写。

与单体系统的区别

在以往传统的企业系统架构中，我们针对一个复杂的业务需求通常使用对象或业务类型来构建一个单体项目。在项目中我们通常将需求分为三个主要部分：数据库、服务端处理、前端展现。在业务发展初期，由于所有的业务逻辑在一个应用中，开发、测试、部署都还比较容易且方便。但是，随着企业的发展，系统为了应对不同的业务需求会不断为该单体项目增加不同的业务模块；同时随着移动端设备的进步，前端展现模块已经不仅仅局限于 Web 的形式，这对于系统后端向前端的支持需要更多的接口模块。单体应用由于面对的业务需求更为宽泛，不断扩大的需求会使得单体应用变得越来越臃肿。单体应用的问题

就逐渐凸显出来，由于单体系统部署在一个进程内，往往我们修改了一个很小的功能，为了部署上线会影响其他功能的运行。并且，单体应用中的这些功能模块的使用场景、并发量、消耗的资源类型都各有不同，对于资源的利用又互相影响，这样使得我们对各个业务模块的系统容量很难给出较为准确的评估。所以，单体系统在初期虽然可以非常方便地进行开发和使用，但是随着系统的发展，维护成本会变得越来越大，且难以控制。

为了解决单体系统变得庞大臃肿之后产生的难以维护的问题，微服务架构诞生了并被大家所关注。我们将系统中的不同功能模块拆分成多个不同的服务，这些服务都能够独立部署和扩展。由于每个服务都运行在自己的进程内，在部署上有稳固的边界，这样每个服务的更新都不会影响其他服务的运行。同时，由于是独立部署的，我们可以更准确地为每个服务评估性能容量，通过配合服务间的协作流程也可以更容易地发现系统的瓶颈位置，以及给出较为准确的系统级性能容量评估。

如何实施微服务

在实施微服务之前，我们必须要知道，微服务虽然有非常多吸引人的优点，但是也因为服务的拆分引发了诸多原本在单体应用中没有的问题。

- **运维的新挑战：**在微服务架构中，运维人员需要维护的进程数量会大大增加。有条不紊地将这些进程编排和组织起来不是一件容易的事，传统的运维人员往往很难适应这样的改变。我们需要运维人员有更多的技能来应对这样的挑战，运维过程需要更多的自动化，这就要求运维人员具备一定的开发能力来编排运维过程并让它们能自动运行起来。
- **接口的一致性：**虽然我们拆分了服务，但是业务逻辑上的依赖并不会消除，只是从单体应用中的代码依赖变为了服务间的通信依赖。而当我们对原有接口进行了一些修改，那么交互方也需要协调这样的改变来进行发布，以保证接口的正确调用。我们需要更完善的接口和版本管理，或是严格地遵循开闭原则。
- **分布式的复杂性：**由于拆分后的各个微服务都是独立部署并运行在各自的进程内，它们只能通过通信来进行协作，所以分布式环境的问题都将是微服务架构系统设计时需要考虑的重要因素，比如网络延迟、分布式事务、异步消息等。

尽管微服务架构有很多缺点和问题，但是其实现的敏捷开发和自动化部署等优点依然被广大优秀架构师和开发者所青睐，所以解决这些问题就是这几年诸多架构大师努力的目标。

在架构师对于一个大型系统架构的设计与实施的过程中，面对环境、资源、团队等各种因素的影响，几乎不会出现完全相同的架构设计。对于微服务架构而言更是如此，由于

并没有一个标准或正式的定义，每位架构师都根据自身理解与实际情况来进行设计，并在发展的过程中不断演化与完善。经过多年的发展，Martin Fowler 在 *Microservices* 一文中，提炼出了微服务架构的九大特性，用于指导大家设计架构。

服务组件化

组件，是一个可以独立更换和升级的单元。就像 PC 中的 CPU、内存、显卡、硬盘一样，独立且可以更换升级而不影响其他单元。

在微服务架构中，需要我们对服务进行组件化分解。服务，是一种进程外的组件，它通过 HTTP 等通信协议进行协作，而不是像传统组件那样以嵌入的方式协同工作。每一个服务都独立开发、部署，可以有效避免一个服务的修改引起整个系统的重新部署。

打一个不恰当的比喻，如果我们的 PC 组件以服务的方式构建，那么只维护主板和一些必要外设之后，计算能力通过一组外部服务实现，我们只需要告诉 PC 从哪个地址来获得计算能力，通过服务定义的计算接口来实现我们使用过程中的计算需求，从而实现 CPU 组件的服务化。这样原本复杂的 PC 服务得到了轻量化的实现，我们甚至只需要更换服务地址就能升级 PC 的计算能力。

按业务组织团队

当决定如何划分微服务时，通常也意味着我们要开始对团队进行重新规划与组织。按以往的方式，我们往往会从技术的层面将团队划分为多个，比如 DBA 团队、运维团队、后端团队、前端团队、设计师团队等。若我们继续按这种方式组织团队来实施微服务架构开发，当有一个服务出现问题需要更改时，可能是一个非常简单的变动，比如对人物描述增加一个字段，这需要从数据存储开始考虑一直到设计和前端，虽然大家的修改都非常小，但这会引起跨团队的时间耗费和预算审批。

在实施微服务架构时，需要采用不同的团队分割方法。由于每一个微服务都是针对特定业务的宽栈或是全栈实现，既要负责数据的持久化存储，又要负责用户的接口定义等各种跨专业领域的职能。因此，面对大型项目的时候，对于微服务团队的拆分更加建议按业务线的方式进行拆分，一方面可以有效减少服务内部修改所产生的内耗；另一方面，团队边界可以变得更为清晰。

做“产品”的态度

在实施微服务架构的团队中，每个小团队都应该以做产品的方式，对其产品的整个生命周期负责。而不是以项目的模式，以完成开发与交付并将成果交接给维护者为最终目标。

开发团队通过了解服务在具体生产环境中的情况，可以增加他们对具体业务的理解，比如，很多时候，一些业务中发生的特殊或异常情况，很可能产品经理都并不知晓，但细心的开发者很容易通过生产环境发现这些特殊的潜在问题或需求。

所以，我们需要用做"产品"的态度来对待每一个微服务，持续关注服务的运作情况，并不断分析以帮助用户来改善业务功能。

智能端点与哑管道

在单体应用中，组件间直接通过函数调用的方式进行交互协作。而在微服务架构中，由于服务不在一个进程中，组件间的通信模式发生了改变，若仅仅将原本在进程内的方法调用改成 RPC 方式的调用，会导致微服务之间产生烦琐的通信，使得系统表现更为糟糕，所以，我们需要更粗粒度的通信协议。

在微服务架构中，通常会使用以下两种服务调用方式：

- 第一种，使用 HTTP 的 RESTful API 或轻量级的消息发送协议，实现信息传递与服务调用的触发。
- 第二种，通过在轻量级消息总线上传递消息，类似 RabbitMQ 等一些提供可靠异步交换的中间件。

> 在极度强调性能的情况下，有些团队会使用二进制的消息发送协议，例如 protobuf。即使是这样，这些系统仍然会呈现出"智能端点和哑管道"的特点，这是为了在易读性与高效性之间取得平衡。当然大多数 Web 应用或企业系统并不需要在这两者间做出选择，能够获得易读性已经是一个极大的胜利了。
>
> ——Martin Fowler

去中心化治理

当我们采用集中化的架构治理方案时，通常在技术平台上都会制定统一的标准，但是每一种技术平台都有其短板，这会导致在碰到短板时，不得不花费大力气去解决，并且可能因为其底层原因解决得不是很好，最终成为系统的瓶颈。

在实施微服务架构时，通过采用轻量级的契约定义接口，使得我们对于服务本身的具体技术平台不再那么敏感，这样整个微服务架构系统中的各个组件就能针对其不同的业务特点选择不同的技术平台，终于不会出现杀鸡用牛刀或是杀牛用指甲钳的尴尬处境了。

> 不是每一个问题都是钉子，不是每一个解决方案都是锤子。

去中心化管理数据

我们在实施微服务架构时，都希望让每一个服务来管理其自有的数据库，这就是数据管理的去中心化。

在去中心化过程中，我们除了将原数据库中的存储内容拆分到新的同平台的其他数据库实例中之外(如把原本存储在 MySQL 中的表拆分后，存储到多个不同的 MySQL 实例中)，也可以将一些具有特殊结构或业务特性的数据存储到一些其他技术的数据库实例中（如把日志信息存储到 MongoDB 中或把用户登录信息存储到 Redis 中）。

虽然数据管理的去中心化可以让数据管理更加细致化，通过采用更合适的技术可让数据存储和性能达到最优。但是，由于数据存储于不同的数据库实例中后，数据一致性也成为微服务架构中亟待解决的问题之一。分布式事务本身的实现难度就非常大，所以在微服务架构中，我们更强调在各服务之间进行“无事务”的调用，而对于数据一致性，只要求数据在最后的处理状态是一致的即可；若在过程中发现错误，通过补偿机制来进行处理，使得错误数据能够达到最终的一致性。

基础设施自动化

近年来云计算服务与容器化技术的不断成熟，运维基础设施的工作变得越来越容易。但是，当我们实施微服务架构时，数据库、应用程序的个头虽然都变小了，但是因为拆分的原因，数量成倍增长。这使得运维人员需要关注的内容也成倍增长，并且操作性任务也会成倍增长，这些问题若没有得到妥善解决，必将成为运维人员的噩梦。

所以，在微服务架构中，务必从一开始就构建起“持续交付”平台来支撑整个实施过程，该平台需要两大内容，缺一不可。

- **自动化测试：**每次部署前的强心剂，尽可能地获得对正在运行的软件的信心。
- **自动化部署：**解放烦琐枯燥的重复操作以及对多环境的配置管理。

容错设计

在单体应用中，一般不存在单个组件故障而其他部件还在运行的情况，通常是一挂全挂。而在微服务架构中，由于服务都运行在独立的进程中，所以存在部分服务出现故障，而其他服务正常运行的情况。比如，当正常运作的服务 B 调用到故障服务 A 时，因故障服务 A 没有返回，线程挂起开始等待，直到超时才能释放，而此时若触发服务 B 调用服务 A 的请求来自服务 C，而服务 C 频繁调用服务 B 时，由于其依赖服务 A，大量线程被挂起等待，最后导致服务 A 也不能正常服务，这时就会出现故障的蔓延。

所以，在微服务架构中，快速检测出故障源并尽可能地自动恢复服务是必须被设计和

考虑的。通常，我们都希望在每个服务中实现监控和日志记录的组件，比如服务状态、断路器状态、吞吐量、网络延迟等关键数据的仪表盘等。

演进式设计

通过上面的几点特征，我们已经能够体会到，要实施一个完美的微服务架构，需要考虑的设计与成本并不小，对于没有足够经验的团队来说，甚至要比单体应用付出更多的代价。

所以，在很多情况下，架构师都会以演进的方式进行系统的构建。在初期，以单体系统的方式来设计和实施，一方面系统体量初期并不会很大，构建和维护成本都不高。另一方面，初期的核心业务在后期通常也不会发生巨大的改变。随着系统的发展或者业务的需要，架构师会将一些经常变动或是有一定时间效应的内容进行微服务处理，并逐渐将原来在单体系统中多变的模块逐步拆分出来，而稳定不太变化的模块就形成一个核心微服务存在于整个架构之中。

为什么选择 Spring Cloud

近几年很多人对于微服务架构的热情非常高，但是回头看“微服务”被提及也有很多年了。无数的架构师和开发者在实际项目中实践该设计理念并为此付出了诸多努力，同时也分享了他们在微服务架构中针对不同应用场景出现的各种问题的各种解决方案和开源框架，其中也不乏国内互联网企业的杰出贡献。

- **服务治理**：阿里巴巴开源的 Dubbo 和当当网在其基础上扩展的 DubboX、Netflix 的 Eureka、Apache 的 Consul 等。
- **分布式配置管理**：百度的 Disconf、Netflix 的 Archaius、360 的 QConf、Spring Cloud 的 Config、淘宝的 Diamond 等。
- **批量任务**：当当网的 Elastic-Job、LinkedIn 的 Azkaban、Spring Cloud 的 Task 等。
- **服务跟踪**：京东的 Hydra、Spring Cloud 的 Sleuth、Twitter 的 Zipkin 等。
- ……

上面列举了一些在实施微服务架构初期，就需要被我们考虑进去的问题，以及针对这些问题的开源解决方案。可以看到国内、国外的技术公司都在贡献着他们的智慧。我们搜索微服务架构的实施方案时会发现，几乎大部分的分享主要以理论或是一个粗轮廓框架为主，整合了来自不同公司或组织的诸多开源框架，并加入针对自身业务的一些优化，所以找不到一个完全相同的架构方案。

前面我们介绍了一些关于微服务的理念以及特性，分析了实施微服务的优点和缺点，

而这些缺点通常就是这些框架出现的源头，大家都是为了解决或弥补业务拆分后所引出的诸多问题来设计出这些解决方案。而当我们作为一个新手，准备实施微服务架构时，为了避免踩前辈们踩过的坑，我们不得不在这些核心问题上做出选择，而选择又是如此之多，这必然会导致在做技术选型的初期，需要花费巨大的调研、分析与实验精力。

Spring Cloud 的出现，可以说是对微服务架构的巨大支持和强有力的技术后盾。它不像我们之前所列举的框架那样，只是解决微服务中的某一个问题，而是一个解决微服务架构实施的综合性解决框架，它整合了诸多被广泛实践和证明过的框架作为实施的基础部件，又在该体系基础上创建了一些非常优秀的边缘组件。

打个不太恰当的比喻：我们自己对各个问题选择框架来实施微服务架构就像在 DIY 电脑一样，我们对各环节的选择自由度很高，但是最终结果很有可能因为一条内存质量不行就点不亮了，总是让人不怎么放心。当然，如果你是一名高手，这些自然都不是问题，然而千军易得、良将难求。而使用 Spring Cloud 来实施就像直接购买品牌机一样，在 Spring 社区的整合之下，做了大量的兼容性测试，保证了其拥有更好的稳定性，如果要在 Spring Cloud 架构下使用非原装组件时，就需要对其基础有足够的了解。

Spring Cloud 也许对很多已经实施微服务并自成体系的团队不具备足够的吸引力，但是对于还未实施微服务或是未成体系的团队，这必将是一个非常有吸引力的框架选择。不论其项目的发展目标，还是 Spring 的强大背景，亦或其极高的社区活跃度，都是未来企业架构师必须了解和接触的重要框架，有一天成为微服务架构的标准解决方案也并非不可能。

Spring Cloud 简介

Spring Cloud 是一个基于 Spring Boot 实现的微服务架构开发工具。它为微服务架构中涉及的配置管理、服务治理、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等操作提供了一种简单的开发方式。

Spring Cloud 包含了多个子项目（针对分布式系统中涉及的多个不同开源产品，还可能会新增），如下所述。

- Spring Cloud Config：配置管理工具，支持使用 Git 存储配置内容，可以使用它实现应用配置的外部化存储，并支持客户端配置信息刷新、加密/解密配置内容等。
- Spring Cloud Netflix：核心组件，对多个 Netflix OSS 开源套件进行整合。
  - Eureka：服务治理组件，包含服务注册中心、服务注册与发现机制的实现。
  - Hystrix：容错管理组件，实现断路器模式，帮助服务依赖中出现的延迟和为故障提供强大的容错能力。
  - Ribbon：客户端负载均衡的服务调用组件。

  - Feign：基于 Ribbon 和 Hystrix 的声明式服务调用组件。
  - Zuul：网关组件，提供智能路由、访问过滤等功能。
  - Archaius：外部化配置组件。
- Spring Cloud Bus：事件、消息总线，用于传播集群中的状态变化或事件，以触发后续的处理，比如用来动态刷新配置等。
- Spring Cloud Cluster：针对 ZooKeeper、Redis、Hazelcast、Consul 的选举算法和通用状态模式的实现。
- Spring Cloud Cloudfoundry：与 Pivotal Cloudfoundry 的整合支持。
- Spring Cloud Consul：服务发现与配置管理工具。
- Spring Cloud Stream：通过 Redis、Rabbit 或者 Kafka 实现的消费微服务，可以通过简单的声明式模型来发送和接收消息。
- Spring Cloud AWS：用于简化整合 Amazon Web Service 的组件。
- Spring Cloud Security：安全工具包，提供在 Zuul 代理中对 OAuth2 客户端请求的中继器。
- Spring Cloud Sleuth：Spring Cloud 应用的分布式跟踪实现，可以完美整合 Zipkin。
- Spring Cloud ZooKeeper：基于 ZooKeeper 的服务发现与配置管理组件。
- Spring Cloud Starters：Spring Cloud 的基础组件，它是基于 Spring Boot 风格项目的基础依赖模块。
- Spring Cloud CLI：用于在 Groovy 中快速创建 Spring Cloud 应用的 Spring Boot CLI 插件。
- ……

本书将对其中一些较为常用的组件进行介绍、分析，并演示其使用方法。

版本说明

当我们通过搜索引擎查找一些 Spring Cloud 的文章或示例时，往往可以在依赖中看到很多不同的版本名字，比如 Angel.SR6、Brixton.SR5 等，为什么 Spring Cloud 没有像其他 Spring 的项目使用类似 1.x.x 的版本命名规则呢？这些版本之间又有什么区别呢？在学习之初，非常有必要弄清楚这些版本的意义和内容，这样才能在我们使用 Spring Cloud 时，指导我们选择更为合适的版本进行架构与开发。

版本名与版本号

由于 Spring Cloud 不像 Spring 社区其他一些项目那样相对独立，它是一个拥有诸多子项目的大型综合项目，可以说是对微服务架构解决方案的综合套件组合，其包含的各个子

项目也都独立进行着内容更新与迭代，各自都维护着自己的发布版本号。因此每一个 Spring Cloud 的版本都会包含多个不同版本的子项目，为了管理每个版本的子项目清单，避免 Spring Cloud 的版本号与其子项目的版本号相混淆，没有采用版本号的方式，而是通过命名的方式。

这些版本的名字采用了伦敦地铁站的名字，根据字母表的顺序来对应版本时间顺序，比如最早的 Release 版本为 Angel，第二个 Release 版本为 Brixton……

经过上面的解释，不难猜出，之前所提到的 Angel.SR6、Brixton.SR5 中的 SR6、SR5 就是版本号了。

当一个版本的 Spring Cloud 项目的发布内容积累到临界点或者一个严重 bug 解决可用后，就会发布一个“service releases”版本，简称 SRX 版本，其中 X 是一个递增的数字，所以 Brixton. SR5 就是 Brixton 的第 5 个 Release 版本。

版本区别

下面列出的是开始编写此书时各版本的版本构成表，我们可以快速查阅当前各版本所包含的子项目，以及各子项目的版本号，以此来决定需要选择哪个版本。

Component	Angel.SR6	Brixton.SR5	Camden.M1	Camden.BUILD-SNAPSHOT
spring-cloud-aws	1.0.4.RELEASE	1.1.1.RELEASE	1.1.1.RELEASE	1.1.2.BUILD-SNAPSHOT
spring-cloud-bus	1.0.3.RELEASE	1.1.1.RELEASE	1.2.0.M1	1.2.0.BUILD-SNAPSHOT
spring-cloud-cli	1.0.6.RELEASE	1.1.5.RELEASE	1.2.0.M1	1.2.0.BUILD-SNAPSHOT
spring-cloud-commons	1.0.5.RELEASE	1.1.1.RELEASE	1.1.1.RELEASE	1.1.2.BUILD-SNAPSHOT
spring-cloud-contract			1.0.0.M2	1.0.0.BUILD-SNAPSHOT
spring-cloud-config	1.0.4.RELEASE	1.1.3.RELEASE	1.2.0.M1	1.2.0.BUILD-SNAPSHOT
spring-cloud-netflix	1.0.7.RELEASE	1.1.5.RELEASE	1.2.0.M1	1.2.0.BUILD-SNAPSHOT
spring-cloud-security	1.0.3.RELEASE	1.1.2.RELEASE	1.1.2.RELEASE	1.1.3.BUILD-SNAPSHOT
spring-cloud-starters	1.0.6.RELEASE			
spring-cloud-cloudfoundry		1.0.0.RELEASE	1.0.0.RELEASE	1.0.1.BUILD-SNAPSHOT
spring-cloud-cluster		1.0.1.RELEASE		
spring-cloud-consul		1.0.2.RELEASE	1.1.0.M1	1.1.0.BUILD-SNAPSHOT
spring-cloud-sleuth		1.0.6.RELEASE	1.0.6.RELEASE	1.0.7.BUILD-SNAPSHOT
spring-cloud-stream		1.0.2.RELEASE	Brooklyn.M1	Brooklyn.BUILD-SNAPSHOT
spring-cloud-zookeeper		1.0.2.RELEASE	1.0.2.RELEASE	1.0.3.BUILD-SNAPSHOT
spring-boot	1.2.8.RELEASE	1.3.7.RELEASE	1.4.0.RELEASE	1.4.0.RELEASE
spring-cloud-task		1.0.2.RELEASE	1.0.2.RELEASE	1.0.3.BUILD-SNAPSHOT

不难看出，最初的 Angel 版本相对来说拥有的子项目较少，Brixton、Camden 则拥有更

全的子项目，所以能提供更多的组件支持。Brixton 的发布子项目更为稳定，Camden 则更为前瞻。

由于 Brixton 版本包含了大部分的 Spring Cloud 子项目，并且均为 Release 版本，所以本文所有示例以及讲解内容的编写均采用 Brixton.SR5 版本，基于 Spring Boot 1.3.7 版本。

注意：在本书完成时，Brixton 版本已经升级到 SR7，本书中的示例均可直接使用 Brixton.SR7 来完成，但是在使用 Brixton 版本的时候需要注意 Spring Boot 的版本，必须使用 1.3.x 版本，而不能使用 1.4.x 版本，否则会出现各种问题。若一定要使用 Spring 1.4.x 版本的话，必须将 Spring Cloud 版本升级到 Camden 版本，目前 Camden 已经发布 Release 版本，所以可以放心使用，最新版本为 Camden.SR3。另外，Camden 版本虽然可以兼容 Brixton 版本中的各种实现方法，但是在升级后读者会发现有一些方法已经被标注为过期，对于这些过期的使用方法，笔者后续会在博客中逐个编写博文来说明这些更新内容。所以，有兴趣的读者可以关注我的博客（http://blog.didispace.com），来持续获取 Spring Cloud 的一些最新动态。

第 2 章

微服务构建：Spring Boot

在展开 Spring Cloud 的微服务架构部署之前，我们先通过本章的内容来了解一下用于构建微服务的基础框架——Spring Boot。对于 Spring Boot 已经有深入了解的读者可以直接跳过本章，进入后续章节学习 Spring Cloud 各个组件的使用。

如果对于 Spring Boot 还不了解的话，建议先读完本章内容之后再继续学习后续关于 Spring Cloud 的内容。由于 Spring Cloud 的构建基于 Spring Boot 实现，在后续的示例中我们将大量使用 Spring Boot 来构建微服务架构中的基础设施以及一些试验中使用的微服务。为了能够辅助后续内容的介绍，确保读者有一定的Spring Boot基础，在这里先对Spring Boot 做一个简单的介绍，以保证读者能够有一定的基础去理解后续介绍的内容并顺利完成后续的一些示例试验。

在这里介绍 Spring Boot 的目的除了它是 Spring Cloud 的基础之外，也由于其自身的各项优点，如自动化配置、快速开发、轻松部署等，非常适合用作微服务架构中各项具体微服务的开发框架。所以我们强烈推荐使用 Spring Boot 来构建微服务，它不仅可以帮助我们快速地构建微服务，还可以轻松简单地整合 Spring Cloud 实现系统服务化，而如果使用了传统的 Spring 构建方式的话，在整合过程中我们还需要做更多的依赖管理工作才能让它们完好地运行起来。

在本章中我们将介绍下面这些与后续介绍有密切联系的内容：

- 如何构建 Spring Boot 项目
- 如何实现 RESTful API 接口
- 如何实现多环境的 Spring Boot 应用配置
- 深入理解 Spring Boot 配置的启动机制
- Spring Boot 应用的监控与管理

更多关于 Spring Boot 的使用细节，读者可以参阅 Spring Boot 的官方文档或是其他资料来进一步学习。

框架简介

对于很多 Spring 框架的初学者来说，经常会因为其繁杂的配置文件而却步。而对于很多老手来说，每次新构建项目总是会重复复制粘贴一些差不多的配置文件这样枯燥乏味的事。作为一名优秀的程序员或架构师，我们总会想尽办法来避免这样的重复劳动，比如，通过 Maven 等构建工具来创建针对不同场景的脚手架工程，在需要新建项目时通过这些脚手架来初始化我们自定义的标准工程，并根据需要做一些简单修改以达到简化原有配置过程的效果。这样的做法虽然减少了工作量，但是这些配置依然大量散布在我们的工程中，大部分情况下我们并不会去修改这些内容，但为什么还要反复出现在我们的工程中呢？实在有些碍眼！

Spring Boot 的出现可以有效改善这类问题，Spring Boot 的宗旨并非要重写 Spring 或是替代 Spring，而是希望通过设计大量的自动化配置等方式来简化 Spring 原有样板化的配置，使得开发者可以快速构建应用。

除了解决配置问题之外，Spring Boot 还通过一系列 Starter POMs 的定义，让我们整合各项功能的时候，不需要在 Maven 的 `pom.xml` 中维护那些错综复杂的依赖关系，而是通过类似模块化的 Starter 模块定义来引用，使得依赖管理工作变得更为简单。

在如今容器化大行其道的时代，Spring Boot 除了可以很好融入 Docker 之外，其自身就支持嵌入式的 Tomcat、Jetty 等容器。所以，通过 Spring Boot 构建的应用不再需要安装 Tomcat，将应用打包成 war，再部署到 Tomcat 这样复杂的构建与部署动作，只需将 Spring Boot 应用打成 jar 包，并通过 `java -jar` 命令直接运行就能启动一个标准化的 Web 应用，这使得 Spring Boot 应用变得非常轻便。

Spring Boot 对于构建、部署等做了这么多的优化，自然不能少了对开发环节的优化。整个 Spring Boot 的生态系统都使用到了 Groovy，很自然的，我们完全可以通过使用 Gradle 和 Groovy 来开发 Spring Boot 应用，比如下面短短的不足 100 个字符的代码，通过编译打包，使用 `java -jar` 命令就能启动一个返回“hello”的 RESTful API。

```
@RestController
class App {

    @RequestMapping("/")
    String home() {
```

```
        "hello"
    }

}
```

说了这么多 Spring Boot 带来的颠覆性框架特性，下面我们就通过后续内容来体验一下使用 Spring Boot 构建微服务的过程，以对 Spring Boot 有一个直观的感受。

快速入门

在本节中，我们将逐步指引读者创建一个 Spring Boot 的基础项目，并且实现一个简单的 RESTful API，通过这个例子对 Spring Boot 有一个初步的了解，并体验其结构简单、开发迅速的特性。

项目构建与解析

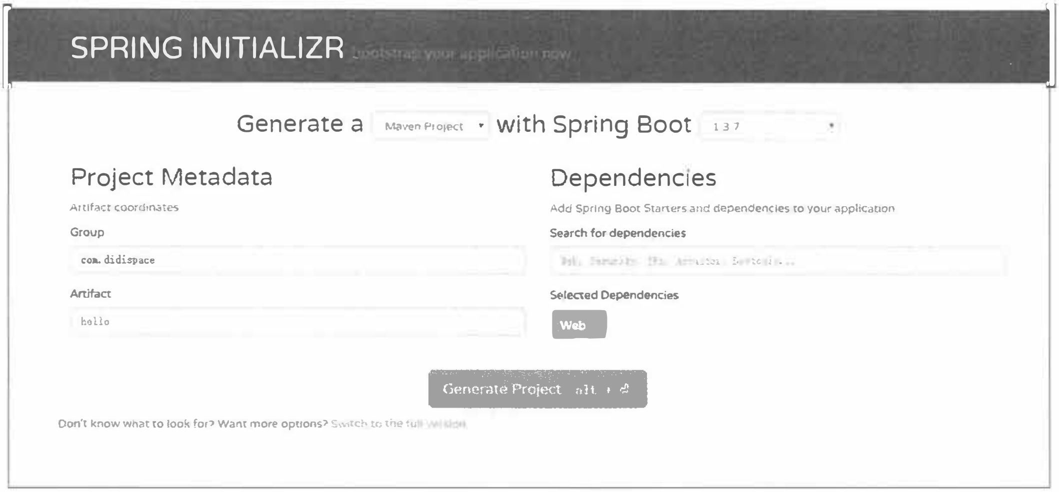
系统及工具版本要求

- Java 7 及以上版本
- Spring Framework 4.2.7 及以上版本
- Maven 3.2 及以上版本 / Gradle 1.12 及以上版本

本书内容均采用 Java 1.8.0_73、Spring Boot 1.3.7 调试通过。

构建 Maven 项目

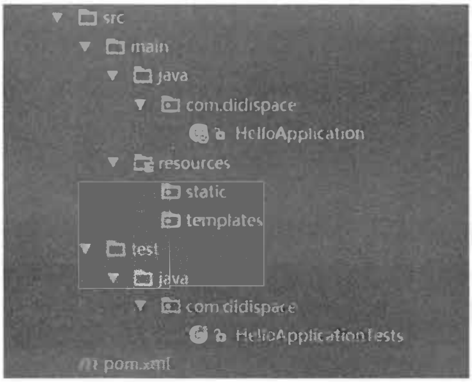
1. 通过官方的 Spring Initializr 工具来产生基础项目。
2. 访问 `http://start.spring.io/`，如下图所示，该页面提供了以 Maven 或 Gradle 构建 Spring Boot 项目的功能。
3. 选择构建工具 `Maven Project`、Spring Boot 版本选择 `1.3.7`，填写 Group 和 Artifact 信息，在 `Search for dependencies` 中可以搜索需要的其他依赖包，这里我们要实现 RESTful API，所以可以添加 `Web` 依赖。
4. 单击 `Generate Project` 按钮下载项目压缩包。
5. 解压项目包，并用 IDE 以 `Maven` 项目导入，以 `IntelliJ IDEA 14` 为例。
6. 从菜单中选择 `File-->New-->Project from Existing Sources...`。
7. 选择解压后的项目文件夹，单击 `OK` 按钮。



8. 单击 `Import project from external model` 并选择 `Maven`，一直单击 `Next` 按钮。
9. 若你的环境中有多个版本的 JDK，选择 `Java SDK` 的时候请选择 `Java 7` 以上的版本。

工程结构解析

在完成了上面的步骤之后，我们就创建了一个最基础的 Spring Boot 工程。



如上图所示，Spring Boot 的基础结构有三大块（具体路径根据用户生成项目时填写的 `Group` 和 `Artifact` 有所差异）。

- `src/main/java`：主程序入口 `HelloApplication`，可以通过直接运行该类来启动 Spring Boot 应用。

- src/main/resources：配置目录，该目录用来存放应用的一些配置信息，比如应用名、服务端口、数据库链接等。由于我们引入了 Web 模块，因此产生了 static 目录与 templates 目录，前者用于存放静态资源，如图片、CSS、JavaScript 等；后者用于存放 Web 页面的模板文件，这里我们主要演示提供 RESTful API，所以这两个目录并不会用到。
- src/test/：单元测试目录，生成的 HelloApplicationTests 通过 JUnit 4 实现，可以直接用运行 Spring Boot 应用的测试。后文中，我们会演示如何在该类中测试 RESTful API。

Maven 配置分析

打开当前工程下的 pom.xml 文件，看看生成的项目都引入了哪些依赖来构建 Spring Boot 工程，内容大致如下所示。

```
    <?xml version="1.0" encoding="UTF-8"?>
    <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

        <modelVersion>4.0.0</modelVersion>
        <groupId>com.didispace</groupId>
        <artifactId>spring-boot-hello</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <packaging>jar</packaging>
        <name>spring-boot-hello</name>
        <description>Demo project for Spring Boot</description>

        <parent>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>1.3.7.RELEASE</version>
            <relativePath/> <!-- lookup parent from repository -->
        </parent>

        <properties>
            <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
            <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
            <java.version>1.8</java.version>
        </properties>

        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
```

```
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

在基础信息部分，groupId 和 artifactId 对应生成项目时页面上输入的内容。另外，我们还可以注意到，打包形式为 jar：<packaging>jar</packaging>，正如我们之前所介绍的，Spring Boot 默认将该 Web 应用打包为 jar 的形式，而非 war 的形式，因为默认的 Web 模块依赖会包含嵌入式的 Tomcat，这样使得我们的应用 jar 自身就具备了提供 Web 服务的能力，后续我们会演示如何启动它。

父项目 parent 配置指定为 spring-boot-starter-parent 的 1.3.7 版本，该父项目中定义了 Spring Boot 版本的基础依赖以及一些默认配置内容，比如，配置文件 application.properties 的位置等。

在项目依赖 dependencies 配置中，包含了下面两项。

- spring-boot-starter-web：全栈 Web 开发模块，包含嵌入式 Tomcat、Spring MVC。
- spring-boot-starter-test：通用测试模块，包含 JUnit、Hamcrest、Mockito。

这里所引用的 web 和 test 模块，在 Spring Boot 生态中被称为 Starter POMs。Starter POMs 是一系列轻便的依赖包，是一套一站式的 Spring 相关技术的解决方案。开发者在使用和整合模块时，不必再去搜寻样例代码中的依赖配置来复制使用，只需要引入对应的模块包即可。比如，开发 Web 应用的时候，就引入 spring-boot-starter-web，希望应用具备访问数据库能力的时候，那就再引入 spring-boot-starter-jdbc 或是更好用的 spring-boot-starter-data-jpa。在使用 Spring Boot 构建应用的时候，各项功能模块的整合不再像传统 Spring 应用的开发方式那样，需要在 pom.xml 中做大量的依赖配置，

而是通过使用 Starter POMs 定义的依赖包，使得功能模块整合变得非常轻巧，易于理解与使用。

Spring Boot 的 Starter POMs 采用 `spring-boot-starter-*`的命名方式，*代表一个特别的应用功能模块，比如这里的 web、test。Spring Boot 工程本身的结构非常简单，大量的学习要点还是将来在对这些 Starter POMs 的使用之上。在本书中，由于主要讲述 Spring Cloud 的微服务组件内容，因此对各个 Spring Boot 的模块内容不做详尽讲解。对于初学者来说，我们也不必一次性地将所有模块的详细用法都掌握牢固，只需了解每个模块能做什么即可。对于本书所使用的 Spring Boot 1.3.7 版本的所有 StarterPOMs 的功能索引可以参见附录 A，可以根据自身的需要来查询能够支持的模块，再去搜索这个模块的使用方法来实现我们的需求。

最后，项目构建的 build 部分，引入了 Spring Boot 的 Maven 插件，该插件非常实用，可以帮助我们方便地启停应用，这样在开发时就不用每次去找主类或是打包成 jar 来运行微服务，只需要通过 `mvn spring-boot:run` 命令就可以快速启动 Spring Boot 应用。

实现 RESTful API

在 Spring Boot 中创建一个 RESTful API 的实现代码同 Spring MVC 应用一样，只是不需要像 Spring MVC 那样先做很多配置，而是像下面这样直接开始编写 Controller 内容：

- 新建 `package`，命名为 `com.didispace.web`，可根据实际的构建情况修改成自己的路径。
- 新建 `HelloController` 类，内容如下所示。

```
@RestController
public class HelloController {

    @RequestMapping("/hello")
    public String index() {
        return "Hello World";
    }

}
```

- 启动该应用，通过浏览器访问 `http://localhost:8080/hello`，我们可以看到返回了预期结果：`Hello World`。

启动 Spring Boot 应用

启动 Spring Boot 应用的方式有很多种：

- 作为一个 Java 应用程序，可以直接通过运行拥有 main 函数的类来启动。
- 在 Maven 配置中，之前提到了 spring-boot 插件，可以使用它来启动，比如执行 `mvn spring-boot:run` 命令，或是直接单击 IDE 中对 Maven 插件的工具，例如 IntelliJ 中的支持：

- 在服务器上部署运行时，通常先使用 `mvn install` 将应用打包成 jar 包，再通过 `java -jar xxx.jar` 来启动应用。

编写单元测试

功能实现之后，我们要养成随手写配套单元测试的习惯，这在微服务架构中尤为重要。通常，我们实施微服务架构的时候，已经实现了前后端分离的项目与架构部署。那么在实现后端服务的时候，单元测试是在开发过程中用来验证代码正确性非常好的手段，并且这些单元测试将会很好地支持我们未来可能会进行的重构。

在 Spring Boot 中实现单元测试同样非常方便，下面我们打开 `src/test/` 下的测试入口 `com.didispace.HelloApplicationTests` 类，编写一个简单的单元测试来模拟 HTTP 请求，测试之前实现的 `/hello` 接口，该接口应返回 `Hello World` 字符串。

具体代码实现如下所示。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = HelloApplication.class)
@WebAppConfiguration
public class HelloApplicationTests {

    private MockMvc mvc;

    @Before
    public void setUp() throws Exception {
        mvc = MockMvcBuilders.standaloneSetup(new HelloController()).build();
    }

    @Test
    public void hello() throws Exception {
        mvc.perform(MockMvcRequestBuilders.get("/hello").accept(MediaType.
APPLICATION_JSON))
```

```
                .andExpect(status().isOk())
                .andExpect(content().string(equalTo("Hello World")));
    }
}
```

代码解析如下。

- `@RunWith(SpringJUnit4ClassRunner.class)`：引入 Spring 对 JUnit4 的支持。
- `@SpringApplicationConfiguration(classes = HelloApplication.class)`：指定 Spring Boot 的启动类。
- `@WebAppConfiguration`：开启 Web 应用的配置，用于模拟 ServletContext。
- `MockMvc` 对象：用于模拟调用 Controller 的接口发起请求，在`@Test` 定义的 hello 测试用例中，perform 函数执行一次请求调用，`accept` 用于执行接收的数据类型，`andExpect` 用于判断接口返回的期望值。
- `@Before`：JUnit 中定义在测试用例`@Test` 内容执行前预加载的内容，这里用来初始化对 HelloController 的模拟。

注意引入下面的静态引用，让 `status`、`content`、`equalTo` 函数可用：

```
    import static org.hamcrest.Matchers.equalTo;
    import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.
content;
    import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.
status;
```

快速入门总结

在本节中，我们通过 Spring 官方的项目构建工具 `Spring Initializr` 生成了一个 Spring Boot 的基础项目，并详细介绍了该基础项目的依赖结构以及 Maven 的 spring-boot 插件，接着在该项目中实现一个输出“Hello World”的 RE STful 接口以及针对该接口的单元测试用例，完成了一个虽然简单，但涵盖了项目构建、服务开发、单元测试的全套开发内容。由于我们在后续使用 Spring Cloud 的时候会构建多个 Spring Boot 的项目来实现微服务架构中的基础设施以及微服务应用，所以通过本节的学习，我们已经具备了通过使用 Spring Boot 来构建简单微服务项目的基本能力。

在下一节中，我们将对后续在 Spring Cloud 组件使用过程中会涉及的配置内容做一些介绍和解释。

配置详解

在上一节中，我们轻松地实现了一个简单的 RESTful API 应用，体验了 Spring Boot 的诸多优点。我们用非常少的代码就成功实现了一个 Web 应用，这是传统 Spring 应用无法办到的。虽然在实现 Controller 时用到的代码是一样的，但是在配置方面，相信大家也注意到了，在上面的例子中，除了 Maven 的配置之外，没有引入任何其他配置。

这就是之前我们提到的，Spring Boot 针对常用的开发场景提供了一系列自动化配置来减少原本复杂而又几乎很少改动的模板化配置内容。但是，我们还是需要了解如何在 Spring Boot 中修改这些自动化的配置内容，以应对一些特殊的场景需求，比如，我们在同一台主机上需要启动多个基于 Spring Boot 的 Web 应用，若不为每个应用指定特别的端口号，那么默认的 8080 端口必将导致冲突。

后续我们在使用 Spring Cloud 的各个组件的时候，其实有大量的工作都会是针对配置文件的。所以我们有必要深入了解一些关于 Spring Boot 中的配置文件的知识，比如配置方式、如何实现多环境配置、配置信息的加载顺序等。

配置文件

在快速入门示例中，我们介绍 Spring Boot 的工程结构时，提到过 `src/main/resources` 目录是 Spring Boot 的配置目录，所以当要为应用创建个性化配置时，应在该目录下进行。

Spring Boot 的默认配置文件位置为 `src/main/resources/application.properties`。关于 Spring Boot 应用的配置内容都可以集中在该文件中，根据我们引入的不同 Starter 模块，可以在这里定义容器端口号、数据库连接信息、日志级别等各种配置信息。比如，我们需要自定义 Web 模块的服务端口号，可以在 `application.properties` 中添加 `server.port=8888` 来指定服务端口为 8888，也可以通过 `spring.application.name=hello` 来指定应用名（该名字在后续 Spring Cloud 中会被注册为服务名）。

Spring Boot 的配置文件除了可以使用传统的 properties 文件之外，还支持现在被广泛推荐使用的 YAML 文件。

> YAML（英语发音为/ˈjæməl/，尾音类似 camel 骆驼）是一个可读性高，用来表达资料序列的格式。YAML 参考了其他多种语言，包括 C 语言、Python、Perl，并从 XML、电子邮件的数据格式（RFC 2822）中获得灵感。Clark Evans 在 2001 年首次发表了这种语言，Ingy döt Net 与 Oren Ben-Kiki 也是这种语言的共同设计者。目前已经有数种编程语言或脚本语言支持（或者说解析）这种语言。YAML

是 YAML Ain't a Markup Language（YAML 不是一种标记语言）的缩写。在开发这种语言时，YAML 的意思其实是：Yet Another Markup Language（仍是一种标记语言），但为了强调这种语言以数据作为中心，而不是以标记语言为重点，而用反向缩略语重新命名。YAML 的语法和其他高阶语言类似，并且可以简单地表达清单、散列表、标量等形态。它使用空白符号缩排和大量依赖外观的特色，特别适合用来表达或编辑数据结构、各种设定文档、文件大纲（例如，许多电子邮件标题格式和 YAML 非常接近）。尽管它比较适合表达阶层式（hierarchical model）的数据结构，不过也有精致的语法可以表示关联性（relational model）的资料。由于 YAML 使用空白符号和分行来分隔资料，使得它特别适合用 grep / Python / Perl / Ruby 操作。其让人最容易上手的特色是巧妙避开各种封闭符号，如引号、各种括号等，这些符号在巢状结构时会变得复杂而难以辨认。

——维基百科

YAML 采用的配置格式不像 properties 的配置那样以单纯的键值对形式来表示，而是以类似大纲的缩进形式来表示。下面是一段 YAML 配置信息：

```
environments:
    dev:
        url: http://dev.bar.com
        name: Developer Setup
    prod:
        url: http://foo.bar.com
        name: My Cool App
```

与其等价的 properties 配置如下所示：

```
environments.dev.url=http://dev.bar.com
environments.dev.name=Developer Setup
environments.prod.url=http://foo.bar.com
environments.prod.name=My Cool App
```

通过 YAML 的配置方式我们可以看到，配置信息利用阶梯化缩进的方式，其结构更为清晰易读，同时配置内容的字符量也得到显著减少。除此之外，YAML 还可以在一个单个文件中通过使用 `spring.profiles` 属性来定义多个不同的环境配置。例如下面的内容，在指定为 test 环境时，`server.port` 将使用 8882 端口；而在 prod 环境中，`server.port` 将使用 8883 端口；如果没有指定环境，`server.port` 将使用 8881 端口。

```
server:
    port: 8881
---
spring:
    profiles: test
```

```
server:
  port: 8882
---
spring:
  profiles: prod
server:
  port: 8883
```

> **注意**　YAML 目前还有一些不足，它无法通过@PropertySource 注解来加载配置。但是，YAML 将属性加载到内存中保存的时候是有序的，所以当配置文件中的信息需要具备顺序含义时，YAML 的配置方式比起 properties 配置文件更有优势。

自定义参数

除了可以在 Spring Boot 的配置文件中设置各个 Starter 模块中预定义的配置属性，也可以在配置文件中定义一些我们需要的自定义属性。比如在 application.properties 中添加：

```
book.name=SpringCloudInAction
book.author=ZhaiYongchao
```

然后，在应用中可以通过@Value 注解来加载这些自定义的参数，比如：

```
@Component
public class Book {

    @Value("${book.name}")
    private String name;
    @Value("${book.author}")
    private String author;

    // 省略getter和setter
}
```

@Value 注解加载属性值的时候可以支持两种表达式来进行配置，如下所示。

- 一种是上面介绍的 PlaceHolder 方式，格式为${...}，大括号内为 PlaceHolder。
- 另一种是使用 SpEL 表达式（Spring Expression Language），格式为#{...}，大括号内为 SpEL 表达式。

参数引用

在 application.properties 中的各个参数之间可以直接通过使用 PlaceHolder 的

方式来进行引用，就像下面的设置：

```
book.name=SpringCloud
book.author=ZhaiYongchao
book.desc=${book.author} is writing《${book.name}》
```

book.desc 参数引用了上文中定义的 book.name 和 book.author 属性，最后该属性的值就是 ZhaiYongchao is writing《SpringCloud》。

使用随机数

在一些特殊情况下，我们希望有些参数每次被加载的时候不是一个固定的值，比如密钥、服务端口等。在 Spr ing Boot 的属性配置文件中，可以通过使用${random}配置来产生随机的 int 值、long 值或者 str ing 字符串，这样我们就可以容易地通过配置随机生成属性，而不是在程序中通过编码来实现这些逻辑。

```
${random}的配置方式主要有以下几种，读者可作为参考使用。
# 随机字符串
com.didispace.blog.value=${random.value}
# 随机int
com.didispace.blog.number=${random.int}
# 随机long
com.didispace.blog.bignumber=${random.long}
# 10 以内的随机数
com.didispace.blog.test1=${random.int(10)}
# 10~20 的随机数
com.didispace.blog.test2=${random.int[10,20]}
```

该配置方式可以设置应用端口等场景，以避免在本地调试时出现端口冲突的麻烦。

命令行参数

回顾一下在本章的“快速入门”小节中，我们还介绍了如何启动 Spr ingBoot 应用，其中提到了使用命令 java -jar 来启动的方式。该命令除了启动应用之外，还可以在命令行中指定应用的参数，比如 java -jar xxx.jar --server.port=8888，直接以命令行的方式来设置 server.port 属性，并将启动应用的端口设为 8888。

在用命令行方式启动 Spr ing Boot 应用时，连续的两个减号 -- 就是对 application.properties 中的属性值进行赋值的标识。所以，java -jar xxx.jar--server.port=8888 命令，等价于在 application.properties 中添加属性 server.port=8888。

通过命令行来修改属性值是 Spr ingBoot 非常重要的一个特性。通过此特性，理论上已

经使得应用的属性在启动前是可变的，所以其中的端口号也好、数据库连接也好，都是可以在应用启动时发生改变的，而不同于以往的 Spring 应用通过 Maven 的 Profile 在编译器中进行不同环境的构建。Spring Boot 的这种方式，可以让应用程序的打包内容贯穿开发、测试以及线上部署，而 Maven 不同 Profile 的方案为每个环境所构建的包，其内容本质上是不同的。但是，如果每个参数都需要通过命令行来指定，这显然也不是一个好的方案，所以下面我们看看如何在 Spring Boot 中实现多环境的配置。

多环境配置

我们在开发应用的时候，通常同一套程序会被应用和安装到几个不同的环境中，比如开发、测试、生产等。其中每个环境的数据库地址、服务器端口等配置都不同，如果在为不同环境打包时都要频繁修改配置文件的话，那必将是个非常烦琐且容易发生错误的事。

对于多环境的配置，各种项目构建工具或是框架的基本思路是一致的，通过配置多份不同环境的配置文件，再通过打包命令指定需要打包的内容之后进行区分打包，Spring Boot 也不例外，或者说实现起来更加简单。

在 Spring Boot 中，多环境配置的文件名需要满足 `application-{profile}.properties` 的格式，其中`{profile}`对应你的环境标识，如下所示。

- `application-dev.properties`：开发环境。
- `application-test.properties`：测试环境。
- `application-prod.properties`：生产环境。

至于具体哪个配置文件会被加载，需要在 `application.properties` 文件中通过 `spring.profiles.active` 属性来设置，其值对应配置文件中的`{profile}`值。如 `spring.profiles.active=test` 就会加载 `application-test.properties` 配置文件内容。

下面，以不同环境配置不同的服务端口为例，进行样例实验。

- 针对各环境新建不同的配置文件 `application-dev.properties`、`application-test.properties`、`application-prod.properties`。
- 在这三个文件中均设置不同的 `server.port` 属性，例如，dev 环境设置为 1111，test 环境设置为 2222，prod 环境设置为 3333。
- `application.properties` 中设置 `spring.profiles.active=dev`，意为默认以 dev 环境设置。
- 测试不同配置的加载。

- 执行 java -jar xxx.jar，可以观察到服务端口被设置为 1111，也就是默认的开发环境（dev）。
- 执行 java -jar xxx.jar --spring.profiles.active=test，可以观察到服务端口被设置为 2222，也就是测试环境的配置（test）。
- 执行 java -jar xxx.jar --spring.profiles.active=prod，可以观察到服务端口被设置为 3333，也就是生产环境的配置（prod）。

按照上面的实验，可以如下总结多环境的配置思路：

- 在 application.properties 中配置通用内容，并设置 spring.profiles.active=dev，以开发环境为默认配置。
- 在 application-{profile}.properties 中配置各个环境不同的内容。
- 通过命令行方式去激活不同环境的配置。

加载顺序

在上面的例子中，我们将 Spring Boot 应用需要的配置内容都放在了项目工程中，已经能够通过 spring.profiles.active 或是通过 Maven 来实现多环境的支持。但是，当团队逐渐壮大，分工越来越细致之后，往往不需要让开发人员知道测试或是生产环境的细节，而是希望由每个环境各自的负责人（QA 或是运维）来集中维护这些信息。那么如果还是以这样的方式存储配置内容，对于不同环境配置的修改就不得不去获取工程内容来修改这些配置内容，当应用非常多的时候就变得非常不方便。同时，配置内容对开发人员都可见，这本身也是一种安全隐患。对此，出现了很多将配置内容外部化的框架和工具，后续将要介绍的 Spring Cloud Config 就是其中之一，为了后续能更好地理解 Spring Cloud Config 的加载机制，我们需要对 Spring Boot 对数据文件的加载机制有一定的了解。

为了能够更合理地重写各属性的值，Spring Boot 使用了下面这种较为特别的属性加载顺序：

1. 在命令行中传入的参数。
2. SPRING_APPLICATION_JSON 中的属性。SPRING_APPLICATION_JSON 是以 JSON 格式配置在系统环境变量中的内容。
3. java:comp/env 中的 JNDI 属性。
4. Java 的系统属性，可以通过 System.getProperties() 获得的内容。
5. 操作系统的环境变量。
6. 通过 random.* 配置的随机属性。

7. 位于当前应用 jar 包之外，针对不同`{profile}`环境的配置文件内容，例如`application-{profile}.properties`或是 YAML 定义的配置文件。
8. 位于当前应用 jar 包之内，针对不同`{profile}`环境的配置文件内容，例如`application-{profile}.properties`或是 YAML 定义的配置文件。
9. 位于当前应用 jar 包之外的`application.properties`和 YAML 配置内容。
10. 位于当前应用 jar 包之内的`application.properties`和 YAML 配置内容。
11. 在`@Configuration`注解修改的类中，通过`@PropertySource`注解定义的属性。
12. 应用默认属性，使用`SpringApplication.setDefaultProperties`定义的内容。

优先级按上面的顺序由高到低，数字越小优先级越高。

可以看到，其中第 7 项和第 9 项都是从应用 jar 包之外读取配置文件，所以，实现外部化配置的原理就是从此切入，为其指定外部配置文件的加载位置来取代 jar 包之内的配置内容。通过这样的实现，我们的工程在配置中就变得非常干净，只需在本地放置开发需要的配置即可，而不用关心其他环境的配置，由其对应环境的负责人去维护即可。

监控与管理

在微服务架构中，我们将原本庞大的单体系统拆分成多个提供不同服务的应用。虽然各个应用的内部逻辑因分解而得以简化，但是由于部署应用的数量成倍增长，使得系统的维护复杂度大大提升。对于运维人员来说，随着应用的不断增多，系统集群中出现故障的频率也变得越来越高，虽然在高可用机制的保护下，个别故障不会影响系统的对外服务，但是这些频繁出现的故障需要被及时发现和处理才能长期保证系统处于健康可用状态。为了能对这些成倍增长的应用做到高效运维，传统的运维方式显然是不合适的，所以我们需要实现一套自动化的监控运维机制，而这套机制的运行基础就是不间断地收集各个微服务应用的各项指标情况，并根据这些基础指标信息来制定监控和预警规则，更进一步甚至做到一些自动化的运维操作等。

为了让运维系统能够获取各个微服务应用的相关指标以及实现一些常规操作控制，我们需要开发一套专门用于植入各个微服务应用的接口供监控系统采集信息。而这些接口往往有很大一部分指标都是类似的，比如环境变量、垃圾收集信息、内存信息、线程池信息等。既然这些信息那么通用，难道就没有一个标准化的实现框架吗？

当我们决定用 Spring Boot 来作为微服务框架时，除了它强大的快速开发功能之外，还因为它在 Starter POMs 中提供了一个特殊依赖模块`spring-boot-starter-actuator`。引

入该模块能够自动为 Spring Boot 构建的应用提供一系列用于监控的端点。同时，Spring Cloud 在实现各个微服务组件的时候，进一步为该模块做了不少扩展，比如，为原生端点增加了更多的指标和度量信息（比如在整合 Eureka 的时候会为/health 端点增加相关的信息），并且根据不同的组件还提供了更多有空的端点（比如，为 API 网关组件 Zuul 提供了/routes 端点来返回路由信息）。

spring-boot-starter-actuator 模块的实现对于实施微服务的中小团队来说，可以有效地省去或大大减少监控系统在采集应用指标时的开发量。当然，它也并不是万能的，有时候也需要对其做一些简单的扩展来帮助我们实现自身系统个性化的监控需求。所以，在本节将详细介绍一些关于 spring-boot-starter-actuator 模块的内容，包括原生提供的端点以及一些常用的扩展和配置方式等。

初识 actuator

下面，我们通过对“快速入门”小节中实现的 Spring Boot 应用增加 spring-boot-starter-actuator 模块功能，来对它有一个直观的认识。

在现有的 Spring Boot 应用中引入该模块非常简单，只需要在 pom.xml 的 dependency 节点中，新增 spring-boot-starter-actuator 的依赖即可，具体如下：

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

增加该依赖之后，重新启动应用。此时，我们可以在控制台中看到如下图所示的输出：

```
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/trace || /trace.json],methods=[GET],produces=[application
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/info || /info.json],methods=[GET],produces=[application/j
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/env/{name:.*}],methods=[GET],produces=[application/json]}
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/env || /env.json],methods=[GET],produces=[application/jso
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/dump || /dump.json],methods=[GET],produces=[application/j
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/autoconfig || /autoconfig.json],methods=[GET],produces=[a
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/health || /health.json],produces=[application/json]}" ont
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/mappings || /mappings.json],methods=[GET],produces=[appli
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/beans || /beans.json],methods=[GET],produces=[application
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/configprops || /configprops.json],methods=[GET],produces=
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/metrics/{name:.*}],methods=[GET],produces=[application/js
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/metrics || /metrics.json],methods=[GET],produces=[applica
```

上图显示了一批端点定义，这些端点并非我们自己在程序中创建的，而是由 spring-boot-starter-actuator 模块根据应用依赖和配置自动创建出来的监控和管理端点。通过这些端点，我们可以实时获取应用的各项监控指标，比如访问/health 端点。我们可以获得如下信息：

```
{
    "status": "UP",
    "diskSpace": {
```

```
        "status": "UP",
        "total": 491270434816,
        "free": 383870214144,
        "threshold": 10485760
    }
}
```

在没有引入其他依赖之前，该端点的内容较为简单，后续我们在使用 Spring Cloud 的各个组件之后，它的返回会变得非常丰富，这些内容将帮助我们制定更为个性化的监控策略。

原生端点

通过在快速入门示例中添加 spring-boot-starter-actuator 模块，我们已经对它有了一个初步的认识。接下来，我们详细介绍一下 spring-boot-starter-actuator 模块中已经实现的一些原生端点。根据端点的作用，可以将原生端点分为以下三大类。

- **应用配置类**：获取应用程序中加载的应用配置、环境变量、自动化配置报告等与 Spring Boot 应用密切相关的配置类信息。
- **度量指标类**：获取应用程序运行过程中用于监控的度量指标，比如内存信息、线程池信息、HTTP 请求统计等。
- **操作控制类**：提供了对应用的关闭等操作类功能。

下面我们来详细了解一下这三类端点都分别可以为我们提供怎样的有用信息和强大功能，以及我们如何去扩展和配置它们。

应用配置类

由于 Spring Boot 为了改善传统 Spring 应用繁杂的配置内容，采用了包扫描和自动化配置的机制来加载原本集中于 XML 文件中的各项内容。虽然这样的做法让我们的代码变得非常简洁，但是整个应用的实例创建和依赖关系等信息都被离散到了各个配置类的注解上，这使我们分析整个应用中资源和实例的各种关系变得非常困难。而这类端点可以帮助我们轻松获取一系列关于 Spring 应用配置内容的详细报告，比如自动化配置的报告、Bean 创建的报告、环境属性的报告等。

- /autoconfig：该端点用来获取应用的自动化配置报告，其中包括所有自动化配置的候选项。同时还列出了每个候选项是否满足自动化配置的各个先决条件。所以，该端点可以帮助我们方便地找到一些自动化配置为什么没有生效的具体原因。该报告内容将自动化配置内容分为以下两部分。
  - positiveMatches 中返回的是条件匹配成功的自动化配置。

- negativeMatches 中返回的是条件匹配不成功的自动化配置。

```
{
    "positiveMatches": { // 条件匹配成功的
        "EndpointWebMvcAutoConfiguration": [
            {
                "condition": "OnClassCondition",
                "message": "@ConditionalOnClass classes found:
javax.servlet.Servlet,org.springframework.web.servlet.DispatcherServlet"
            },
            {
                "condition": "OnWebApplicationCondition",
                "message": "found web application StandardServletEnvironment"
            }
        ],
        ...
    },
    "negativeMatches": { // 条件匹配不成功的
        "HealthIndicatorAutoConfiguration.DataSourcesHealthIndicatorConfiguration": [
            {
                "condition": "OnClassCondition",
                "message": "required @ConditionalOnClass classes not found:
org.springframework.jdbc.core.JdbcTemplate"
            }
        ],
        ...
    }
}
```

从如上示例中我们可以看到，每个自动化配置候选项中都有一系列的条件，比如上面没有成功匹配的 HealthIndicatorAutoConfiguration.DataSourcesHealthIndicatorConfiguration 配置，它的先决条件是需要在工程中包含 org.springframework.jdbc.core.JdbcTemplate 类，由于我们没有引入相关的依赖，它就不会执行自动化配置内容。所以，当我们发现有一些期望的配置没有生效时，就可以通过该端点来查看没有生效的具体原因。

- /beans：该端点用来获取应用上下文中创建的所有 Bean。

```
[
    {
        "context": "hello:dev:8881",
        "parent": null,
        "beans": [
            {
                "bean": "org.springframework.boot.autoconfigure.web.
```

```
DispatcherServletAutoConfiguration$DispatcherServletConfiguration",
                    "scope": "singleton",
                    "type": "org.springframework.boot.autoconfigure.web.
DispatcherServletAutoConfiguration$DispatcherServletConfiguration$$EnhancerBySpring
CGLIB$$3440282b",
                    "resource": "null",
                    "dependencies": [
                        "serverProperties",
                        "spring.mvc.CONFIGURATION_PROPERTIES",
                        "multipartConfigElement"
                    ]
                },
                {
                    "bean": "dispatcherServlet",
                    "scope": "singleton",
                    "type": "org.springframework.web.servlet.DispatcherServlet",
                    "resource": "class path resource
[org/springframework/boot/autoconfigure/web/DispatcherServletAutoConfiguration$Disp
atcherServletConfiguration.class]",
                    "dependencies": []
                }
            ]
        }
    ]
```

在如上示例中，我们可以看到在每个 Bean 中都包含了下面这些信息。

- bean：Bean 的名称。
- scope：Bean 的作用域。
- type：Bean 的 Java 类型。
- resource：class 文件的具体路径。
- dependencies：依赖的 Bean 名称。

- /configprops：该端点用来获取应用中配置的属性信息报告。从下面该端点返回示例的片段中，我们看到返回了关于该短信的配置信息，`prefix` 属性代表了属性的配置前缀，`properties` 代表了各个属性的名称和值。所以，我们可以通过该报告来看到各个属性的配置路径，比如我们要关闭该端点，就可以通过使用 `endpoints.configprops.enabled=false` 来完成设置。

```
{
    "configurationPropertiesReportEndpoint": {
        "prefix": "endpoints.configprops",
        "properties": {
            "id": "configprops",
            "sensitive": true,
```

```
                "enabled": true
            }
        },
        ...
    }
```

- /env：该端点与/configprops 不同，它用来获取应用所有可用的环境属性报告。包括环境变量、JVM 属性、应用的配置属性、命令行中的参数。从下面该端点返回的示例片段中，可以看到它不仅返回了应用的配置属性，还返回了系统属性、环境变量等丰富的配置信息，其中还包括了应用还没有使用的配置，所以它可以帮助我们方便地看到当前应用可以加载的配置信息，并配合@ConfigurationProperties 注解将它们引入到我们的应用程序中来进行使用。另外，为了配置属性的安全，对于一些类似密码等敏感信息，该端点都会进行隐私保护，但是我们需要让属性名中包含 password、secret、key 这些关键词，这样该端点在返回它们的时候会使用*来替代实际的属性值。

```
{
  "profiles": [
    "dev"
  ],
  "server.ports": {
    "local.server.port": 8881
  },
  "servletContextInitParams": {

  },
  "systemProperties": {
     "idea.version": "2016.1.3",
     "java.runtime.name": "Java(TM) SE Runtime Environment",
     "sun.boot.library.path": "C:\\Program Files\\Java\\jdk1.8.0_91\\jre\\bin",
     "java.vm.version": "25.91-b15",
     "java.vm.vendor": "Oracle Corporation",
     ...
  },
  "systemEnvironment": {
     "configsetroot": "C:\\WINDOWS\\ConfigSetRoot",
     "RABBITMQ_BASE": "E:\\tools\\rabbitmq",
     ...
  },
  "applicationConfig: [classpath:/application-dev.properties]": {
     "server.port": "8881"
  },
  "applicationConfig: [classpath:/application.properties]": {
     "server.port": "8885",
```

```
            "spring.profiles.active": "dev",
            "info.app.name": "spring-boot-hello",
            "info.app.version": "v1.0.0",
            "spring.application.name": "hello"
        }
    }
```

- /mappings：该端点用来返回所有 Spring MVC 的控制器映射关系报告。从下面的示例片段中，我们可以看到该报告的信息与我们在启用 Spring MVC 的 Web 应用时输出的日志信息类似，其中 bean 属性标识了该映射关系的请求处理器，method 属性标识了该映射关系的具体处理类和处理函数。

```
{
    "/webjars/**": {
        "bean": "resourceHandlerMapping"
    },
    "/**": {
        "bean": "resourceHandlerMapping"
    },
    "/**/favicon.ico": {
        "bean": "faviconHandlerMapping"
    },
    "{[/hello]}": {
        "bean": "requestMappingHandlerMapping",
        "method": "public java.lang.String com.didispace.web.HelloController.index()"
    },
    "{[/mappings || /mappings.json],methods=[GET],produces=[application/json]}": {
        "bean": "endpointHandlerMapping",
        "method": "public java.lang.Object
org.springframework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invoke()"
    },
    ...
}
```

- /info：该端点用来返回一些应用自定义的信息。默认情况下，该端点只会返回一个空的 JSON 内容。我们可以在 application.properties 配置文件中通过 info 前缀来设置一些属性，比如下面这样：

```
info.app.name=spring-boot-hello
info.app.version=v1.0.0
```

再访问/info 端点，我们可以得到下面的返回报告，其中就包含了上面我们在应用中自定义的两个参数。

```
{
    "app": {
```

```
        "name": "spring-boot-hello",
        "version": "v1.0.0"
    }
}
```

度量指标类

上面我们所介绍的应用配置类端点所提供的信息报告在应用启动的时候就已经基本确定了其返回内容，可以说是一个静态报告。而度量指标类端点提供的报告内容则是动态变化的，这些端点提供了应用程序在运行过程中的一些快照信息，比如内存使用情况、HTTP请求统计、外部资源指标等。这些端点对于我们构建微服务架构中的监控系统非常有帮助，由于 Spring Boot 应用自身实现了这些端点，所以我们可以很方便地利用它们来收集我们想要的信息，以制定出各种自动化策略。下面，我们就来分别看看这些强大的端点功能。

- /metrics：该端点用来返回当前应用的各类重要度量指标，比如内存信息、线程信息、垃圾回收信息等。

```
{
  "mem": 541305,
  "mem.free": 317864,
  "processors": 8,
  "instance.uptime": 33376471,
  "uptime": 33385352,
  "systemload.average": -1,
  "heap.committed": 476672,
  "heap.init": 262144,
  "heap.used": 158807,
  "heap": 3701248,
  "nonheap.committed": 65856,
  "nonheap.init": 2496,
  "nonheap.used": 64633,
  "nonheap": 0,
  "threads.peak": 22,
  "threads.daemon": 20,
  "threads.totalStarted": 26,
  "threads": 22,
  "classes": 7669,
  "classes.loaded": 7669,
  "classes.unloaded": 0,
  "gc.ps_scavenge.count": 7,
  "gc.ps_scavenge.time": 118,
  "gc.ps_marksweep.count": 2,
  "gc.ps_marksweep.time": 234,
  "httpsessions.max": -1,
  "httpsessions.active": 0,
```

```
    "gauge.response.beans": 55,
    "gauge.response.env": 10,
    "gauge.response.hello": 5,
    "gauge.response.metrics": 4,
    "gauge.response.configprops": 153,
    "gauge.response.star-star": 5,
    "counter.status.200.beans": 1,
    "counter.status.200.metrics": 3,
    "counter.status.200.configprops": 1,
    "counter.status.404.star-star": 2,
    "counter.status.200.hello": 11,
    "counter.status.200.env": 1
}
```

从上面的示例中，我们看到有如下这些重要的度量值。

- 系统信息：包括处理器数量 processors、运行时间 uptime 和 instance.uptime、系统平均负载 systemload.average。
- mem.*：内存概要信息，包括分配给应用的总内存数量以及当前空闲的内存数量。这些信息来自 java.lang.Runtime。
- heap.*：堆内存使用情况。这些信息来自 java.lang.management.MemoryMXBean 接口中 getHeapMemoryUsage 方法获取的 java.lang.management.MemoryUsage。
- nonheap.*：非堆内存使用情况。这些信息来自 java.lang.management.MemoryMXBean 接口中 getNonHeapMemoryUsage 方法获取的 java.lang.management.MemoryUsage。
- threads.*：线程使用情况，包括线程数、守护线程数(daemon)、线程峰值(peak)等，这些数据均来自 java.lang.management.ThreadMXBean。
- classes.*：应用加载和卸载的类统计。这些数据均来自 java.lang.management.ClassLoadingMXBean。
- gc.*：垃圾收集器的详细信息，包括垃圾回收次数 gc.ps_scavenge.count、垃圾回收消耗时间 gc.ps_scavenge.time、标记-清除算法的次数 gc.ps_marksweep.count、标记-清除算法的消耗时间 gc.ps_marksweep.time。这些数据均来自 java.lang.management.GarbageCollectorMXBean。
- httpsessions.*：Tomcat 容器的会话使用情况。包括最大会话数 httpsessions.max 和活跃会话数 httpsessions.active。该度量指标信息仅在引入嵌入式 Tomcat 作为应用容器的时候才会提供。

- gauge.*：HTTP 请求的性能指标之一，它主要用来反映一个绝对数值。比如上面示例中的 gauge.response.hello: 5，它表示上一次 hello 请求的延迟时间为 5 毫秒。
- counter.*：HTTP 请求的性能指标之一，它主要作为计数器来使用，记录了增加量和减少量。上述示例中的 counter.status.200.hello: 11，它代表了 hello 请求返回 200 状态的次数为 11。

对于 gauge.*和 counter.*的统计，这里有一个特殊的内容请求 star-star，它代表了对静态资源的访问。这两类度量指标非常有用，我们不仅可以使用它默认的统计指标，还可以在程序中轻松地增加自定义统计值。只需要通过注入 org.springframework.boot.actuate.metrics.CounterService 和 org.springframework.boot.actuate.metrics.GaugeService 来实现自定义的统计指标信息。比如我们可以像下面这样自定义实现对 hello 接口的访问次数统计。

```
@RestController
public class HelloController {

    @Autowired
    private CounterService counterService;

    @RequestMapping("/hello")
    public String greet() {
        counterService.increment("didispace.hello.count");
        return "";
    }
}
```

/metrics 端点可以提供应用运行状态的完整度量指标报告，这项功能非常实用，但是对于监控系统中的各项监控功能，它们的监控内容、数据收集频率都有所不同，如果每次都通过全量获取报告的方式来收集，略显粗暴。所以，我们还可以通过 /metrics/{name} 接口来更细粒度地获取度量信息，比如可以通过访问 /metrics/mem.free 来获取当前可用内存数量。

- /health：该端点在一开始的示例中我们已经使用过了，它用来获取应用的各类健康指标信息。在 spring-boot-starter-actuator 模块中自带实现了一些常用资源的健康指标检测器。这些检测器都通过 HealthIndicator 接口实现，并且会根据依赖关系的引入实现自动化装配，比如下面列出的这些。

检测器	功能
DiskSpaceHealthIndicator	低磁盘空间检测
DataSourceHealthIndicator	检测 DataSource 的连接是否可用
MongoHealthIndicator	检测 Mongo 数据库是否可用
RabbitHealthIndicator	检测 Rabbit 服务器是否可用
RedisHealthIndicator	检测 Redis 服务器是否可用
SolrHealthIndicator	检测 Solr 服务器是否可用

有时候，我们可能还会用到一些 Spring Boot 的 Starter POMs 中还没有封装的产品来进行开发，比如，当使用 RocketMQ 作为消息代理时，由于没有自动化配置的检测器，所以需要自己来实现一个用来采集健康信息的检测器。我们可以在 Spring Boot 的应用中，为 org.springframework.boot.actuate.health.HealthIndicator 接口实现一个对 RocketMQ 的检测器类：

```
@Component
public class RocketMQHealthIndicator implements HealthIndicator {
    @Override
    public Health health() {
        int errorCode = check();
        if (errorCode != 0) {
          return Health.down().withDetail("Error Code", errorCode).build();
        }
        return Health.up().build();
    }

    private int check() {
        // 对监控对象的检测操作
    }
}
```

通过重写 health() 函数可实现健康检查，在返回的 Heath 对象中，共有两项内容，一个是状态信息，除了该示例中的 UP 与 DOWN 之外，还有 UNKNOWN 和 OUT_OF_SERVICE，可以根据需要来实现返回；还有一个详细信息，采用 Map 的方式存储，在这里通过 withDetail 函数，注入了一个 Error Code 信息，我们也可以填入其他信息，比如，检测对象的 IP 地址、端口等。

重新启动应用，并访问/health 接口，我们在返回的 JSON 字符串中，将会包含如下信息：

```
"rocketMQ": {
  "status": "UP"
}
```

- /dump：该端点用来暴露程序运行中的线程信息。它使用 `java.lang.management.ThreadMXBean` 的 `dumpAllThreads` 方法来返回所有含有同步信息的活动线程详情。
- /trace：该端点用来返回基本的 HTTP 跟踪信息。默认情况下，跟踪信息的存储采用 `org.springframework.boot.actuate.trace.InMemoryTraceRepository` 实现的内存方式，始终保留最近的 100 条请求记录。它记录的内容格式如下所示：

```
[
    {
        "timestamp": 1482570022463,
        "info": {
            "method": "GET",
            "path": "/metrics/mem",
            "headers": {
                "request": {
                    "host": "localhost:8881",
                    "connection": "keep-alive",
                    "cache-control": "no-cache",
                    "user-agent": "Mozilla/5.0 (Windows NT 10.0; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.143 Safari/537.36",
                    "postman-token": "9817ea4d-ad9d-b2fc-7685-9dff1a1bc193",
                    "accept": "*/*",
                    "accept-encoding": "gzip, deflate, sdch",
                    "accept-language": "zh-CN,zh;q=0.8"
                },
                "response": {
                    "X-Application-Context": "hello:dev:8881",
                    "Content-Type": "application/json;charset=UTF-8",
                    "Transfer-Encoding": "chunked",
                    "Date": "Sat, 24 Dec 2016 09:00:22 GMT",
                    "status": "200"
                }
            }
        }
    },
    ...
]
```

操作控制类

仔细的读者可能会发现，我们在“初识 actuator”小节中的示例的控制台中输出的所有监控端点，已经在介绍应用配置类端点和度量指标类端点时都讲解完了。那么还有哪些是操作控制类端点呢？实际上，由于之前介绍的所有端点都是用来反映应用自身的属性或是

运行中的状态，相对于操作控制类端点没有那么敏感，所以它们都是默认启用的。而操作控制类端点拥有更强大的控制能力，如果要使用它们的话，需要通过属性来配置开启操作。

在原生端点中，只提供了一个用来关闭应用的端点：/shutdown（在后续我们引入了Eureka之后，会引入更多控制端点）。可以通过如下配置开启它：

```
endpoints.shutdown.enabled=true
```

在配置了上述属性之后，只需要访问该应用的/shutdown端点就能实现关闭该应用的远程操作。由于开放关闭应用的操作本身是一件非常危险的事，所以真正在线上使用的时候，需要对其加入一定的保护机制，比如定制 actuator 的端点路径、整合 Spring Security 进行安全校验等。

小结

本章我们通过构建一个最基本的Spring Boot工程，让大家对其有了一个最直观的感受。同时，也为后续构建各类 Spring Cloud 组件和微服务应用做了一些基础准备工作。另外，我们对 Spring Boot 中的配置原理以及监控管理做了深入的介绍，因为这些内容将在后续的介绍中有所涉及，并且它们有助于理解 Spring Cloud 组件的运行原理。更多关于使用 Spring Boot 开发微服务应用的内容，我们不在本书中详细介绍，读者可参阅官方文档或其他书籍资料来学习。

第 3 章

服务治理：Spring Cloud Eureka

Spring Cloud Eureka 是 Spring Cloud Netflix 微服务套件中的一部分，它基于 Netflix Eureka 做了二次封装，主要负责完成微服务架构中的服务治理功能。Spring Cloud 通过为 Eureka 增加了 Spring Boot 风格的自动化配置，我们只需通过简单引入依赖和注解配置就能让 Spring Boot 构建的微服务应用轻松地与 Eureka 服务治理体系进行整合。

在本章中，我们将指引读者学习下面这些核心内容，并构建起用于服务治理的基础设施。

- 构建服务注册中心
- 服务注册与服务发现
- Eureka 的基础架构
- Eureka 的服务治理机制
- Eureka 的配置

服务治理

服务治理可以说是微服务架构中最为核心和基础的模块，它主要用来实现各个微服务实例的自动化注册与发现。为什么我们在微服务架构中那么需要服务治理模块呢？微服务系统没有它会有什么不好的地方吗？

在最初开始构建微服务系统的时候可能服务并不多，我们可以通过做一些静态配置来完成服务的调用。比如，有两个服务 A 和 B，其中服务 A 需要调用服务 B 来完成一个业务操作时，为了实现服务 B 的高可用，不论采用服务端负载均衡还是客户端负载均衡，都需要手工维护服务 B 的具体实例清单。但是随着业务的发展，系统功能越来越复杂，相应的微服务应用也不断增加，我们的静态配置就会变得越来越难以维护。并且面对不断发展的

业务，我们的集群规模、服务的位置、服务的命名等都有可能发生变化，如果还是通过手工维护的方式，那么极易发生错误或是命名冲突等问题。同时，对于这类静态内容的维护也必将消耗大量的人力。

为了解决微服务架构中的服务实例维护问题，产生了大量的服务治理框架和产品。这些框架和产品的实现都围绕着服务注册与服务发现机制来完成对微服务应用实例的自动化管理。

- **服务注册：**在服务治理框架中，通常都会构建一个注册中心，每个服务单元向注册中心登记自己提供的服务，将主机与端口号、版本号、通信协议等一些附加信息告知注册中心，注册中心按服务名分类组织服务清单。比如，我们有两个提供服务 A 的进程分别运行于 `192.168.0.100:8000` 和 `192.168.0.101:8000` 位置上，另外还有三个提供服务 B 的进程分别运行于 `192.168.0.100:9000`、`192.168.0.101:9000`、`192.168.0.102:9000` 位置上。当这些进程均启动，并向注册中心注册自己的服务之后，注册中心就会维护类似下面的一个服务清单。另外，服务注册中心还需要以心跳的方式去监测清单中的服务是否可用，若不可用需要从服务清单中剔除，达到排除故障服务的效果。

服务名	位置
服务 A	192.168.0.100:8000、192.168.0.101:8000
服务 B	192.168.0.100:9000、192.168.0.101:9000、192.168.0.102:9000

- **服务发现：**由于在服务治理框架下运作，服务间的调用不再通过指定具体的实例地址来实现，而是通过向服务名发起请求调用实现。所以，服务调用方在调用服务提供方接口的时候，并不知道具体的服务实例位置。因此，调用方需要向服务注册中心咨询服务，并获取所有服务的实例清单，以实现对具体服务实例的访问。比如，现有服务 C 希望调用服务 A，服务 C 就需要向注册中心发起咨询服务请求，服务注册中心就会将服务 A 的位置清单返回给服务 C，如按上例服务 A 的情况，C 便获得了服务 A 的两个可用位置 `192.168.0.100:8000` 和 `192.168.0.101:8000`。当服务 C 要发起调用的时候，便从该清单中以某种轮询策略取出一个位置来进行服务调用，这就是后续我们将会介绍的客户端负载均衡。这里我们只是列举了一种简单的服务治理逻辑，以方便理解服务治理框架的基本运行思路。实际的框架为了性能等因素，不会采用每次都向服务注册中心获取服务的方式，并且不同的应用场景在缓存和服务剔除等机制上也会有一些不同的实现策略。

Netflix Eureka

Spring Cloud Eureka，使用 Netflix Eureka 来实现服务注册与发现，它既包含了服务端

组件，也包含了客户端组件，并且服务端与客户端均采用 Java 编写，所以 Eureka 主要适用于通过 Java 实现的分布式系统，或是与 JVM 兼容语言构建的系统。但是，由于 Eureka 服务端的服务治理机制提供了完备的 RESTful API，所以它也支持将非 Java 语言构建的微服务应用纳入 Eureka 的服务治理体系中来。只是在使用其他语言平台的时候，需要自己来实现 Eureka 的客户端程序。不过庆幸的是，在目前几个较为流行的开发平台上，都已经有了一些针对 Eureka 注册中心的客户端实现框架，比如.NET 平台的 Steeltoe、Node.js 的 eureka-js-client 等。

Eureka 服务端，我们也称为服务注册中心。它同其他服务注册中心一样，支持高可用配置。它依托于强一致性提供良好的服务实例可用性，可以应对多种不同的故障场景。如果 Eureka 以集群模式部署，当集群中有分片出现故障时，那么 Eureka 就转入自我保护模式。它允许在分片故障期间继续提供服务的发现和注册，当故障分片恢复运行时，集群中的其他分片会把它们的状态再次同步回来。以在 AWS 上的实践为例，Netflix 推荐每个可用的区域运行一个 Eureka 服务端，通过它来形成集群。不同可用区域的服务注册中心通过异步模式互相复制各自的状态，这意味着在任意给定的时间点每个实例关于所有服务的状态是有细微差别的。

Eureka 客户端，主要处理服务的注册与发现。客户端服务通过注解和参数配置的方式，嵌入在客户端应用程序的代码中，在应用程序运行时，Eureka 客户端向注册中心注册自身提供的服务并周期性地发送心跳来更新它的服务租约。同时，它也能从服务端查询当前注册的服务信息并把它们缓存到本地并周期性地刷新服务状态。

下面我们来构建一些简单示例，学习如何使用 Eureka 构建注册中心以及进行注册与发现服务。

搭建服务注册中心

首先，创建一个基础的 Spring Boot 工程，命名为 `eureka-server`，并在 `pom.xml` 中引入必要的依赖内容，代码如下：

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.3.7.RELEASE</version>
   <relativePath/>
</parent>

<dependencies>
   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka-server</artifactId>
```

```
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Brixton.SR5</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

通过@EnableEurekaServer 注解启动一个服务注册中心提供给其他应用进行对话。这一步非常简单，只需在一个普通的 Spring Boot 应用中添加这个注解就能开启此功能，比如下面的例子：

```
@EnableEurekaServer
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

在默认设置下，该服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为，只需在 application.properties 中增加如下配置：

```
server.port=1111

eureka.instance.hostname=localhost
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${serve
r.port}/eureka/
```

由于后续内容也都会在本地运行，为了与后续要进行注册的服务区分，这里将服务注册中心的端口通过 server.port 属性设置为 1111。

- eureka.client.register-with-eureka：由于该应用为注册中心，所以设置为 false，代表不向注册中心注册自己。

- eureka.client.fetch-registry：由于注册中心的职责就是维护服务实例，它并不需要去检索服务，所以也设置为 false。

在完成了上面的配置后，启动应用并访问 http://localhost:1111/。可以看到如下图所示的 Eureka 信息面板，其中 Instances currently registered with Eureka 栏是空的，说明该注册中心还没有注册任何服务。

注册服务提供者

在完成了服务注册中心的搭建之后，接下来我们尝试将一个既有的 Spring Boot 应用加入 Eureka 的服务治理体系中去。

可以使用上一章中实现的快速入门工程来进行改造，将其作为一个微服务应用向服务注册中心发布自己。首先，修改 pom.xml，增加 Spring Cloud Eureka 模块的依赖，具体代码如下所示：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
```

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Brixton.SR5</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

接着，改造/hello 请求处理接口，通过注入 DiscoveryClient 对象，在日志中打印出服务的相关内容。

```
@RestController
public class HelloController {

    private final Logger logger = Logger.getLogger(getClass());

    @Autowired
    private DiscoveryClient client;

    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    public String index() {
        ServiceInstance instance = client.getLocalServiceInstance();
        logger.info("/hello, host:" + instance.getHost() + ", service_id:" +
instance.getServiceId());
        return "Hello World";
    }

}
```

然后，在主类中通过加上@EnableDiscoveryClient 注解，激活 Eureka 中的 DiscoveryClient 实现（自动化配置，创建 DiscoveryClient 接口针对 Eureka 客户端的 EurekaDiscoveryClient 实例），才能实现上述 Controller 中对服务信息的输出。

```
@EnableDiscoveryClient
@SpringBootApplication
public class HelloApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(HelloApplication.class, args);
    }

}
```

最后，我们需要在 application.properties 配置文件中，通过 spring.application.name 属性来为服务命名，比如命名为 hello-service。再通过 eureka.client.serviceUrl.defaultZone 属性来指定服务注册中心的地址，这里我们指定为之前构建的服务注册中心地址，完整配置如下所示：

```
spring.application.name=hello-service

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

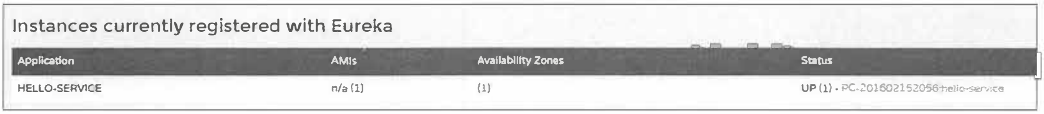
下面我们分别启动服务注册中心以及这里改造后的 hello-service 服务。在 hello-service 服务控制台中，Tomcat 启动之后，com.netflix.discovery.DiscoveryClient 对象打印了该服务的注册信息，表示服务注册成功。

```
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
c.n.e.EurekaDiscoveryClientConfiguration : Updating port to 8080
com.didispace.HelloApplication           : Started HelloApplication in 7.218
seconds (JVM running for 11.646)
com.netflix.discovery.DiscoveryClient    : DiscoveryClient_HELLO-SERVICE/PC-
201602152056:hello-service-registration status: 204
```

而此时在服务注册中心的控制台中，可以看到类似下面的输出，名为 hello-service 的服务被注册成功了。

```
c.n.e.registry.AbstractInstanceRegistry  : Registered instance
HELLO-SERVICE/PC-201602152056:hello-service with status UP (replication=true)
```

我们也可以通过访问 Eureka 的信息面板，在 Instances currently registered with Eureka 一栏中看到服务的注册信息。



通过访问 http://localhost:8080/hello，直接向该服务发起请求，在控制台中可以看到如下输出：

```
com.didispace.web.HelloController        : /hello, host:PC-201602152056,
service_id:hello-service
```

这些输出内容就是之前我们在 HelloController 中注入的 DiscoveryClient 接口对象，从服务注册中心获取的服务相关信息。

高可用注册中心

在微服务架构这样的分布式环境中，我们需要充分考虑发生故障的情况，所以在生产环境中必须对各个组件进行高可用部署，对于微服务如此，对于服务注册中心也一样。但是到本节为止，我们一直都在使用单节点的服务注册中心，这在生产环境中显然并不合适，我们需要构建高可用的服务注册中心以增强系统的可用性。

Eureka Server 的设计一开始就考虑了高可用问题，在 Eureka 的服务治理设计中，所有节点即是服务提供方，也是服务消费方，服务注册中心也不例外。是否还记得在单节点的配置中，我们设置过下面这两个参数，让服务注册中心不注册自己：

```
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
```

Eureka Server 的高可用实际上就是将自己作为服务向其他服务注册中心注册自己，这样就可以形成一组互相注册的服务注册中心，以实现服务清单的互相同步，达到高可用的效果。下面我们就来尝试搭建高可用服务注册中心的集群。可以在本章第 1 节中实现的服务注册中心的基础之上进行扩展，构建一个双节点的服务注册中心集群。

- 创建 `application-peer1.properties`，作为 peer1 服务中心的配置，并将 serviceUrl 指向 peer2：

```
spring.application.name=eureka-server
server.port=1111

eureka.instance.hostname=peer1
eureka.client.serviceUrl.defaultZone=http://peer2:1112/eureka/
```

- 创建 `application-peer2.properties`，作为 peer2 服务中心的配置，并将 serviceUrl 指向 peer1：

```
spring.application.name=eureka-server
server.port=1112

eureka.instance.hostname=peer2
eureka.client.serviceUrl.defaultZone=http://peer1:1111/eureka/
```

- 在`/etc/hosts` 文件中添加对 peer1 和 peer2 的转换，让上面配置的 host 形式的 serviceUrl 能在本地正确访问到；Windows 系统路径为 `C:\Windows\System32\drivers\etc\hosts`。

```
127.0.0.1 peer1
127.0.0.1 peer2
```

- 通过 `spring.profiles.active` 属性来分别启动 peer1 和 peer2：

```
java -jar eureka-server-1.0.0.jar --spring.profiles.active=peer1
java -jar eureka-server-1.0.0.jar --spring.profiles.active=peer2
```

此时访问 peer1 的注册中心 `http://localhost:1111/`，如下图所示，我们可以看到，`registered-replicas` 中已经有 peer2 节点的 eureka-server 了。同样的，我们访问 peer2 的注册中心 `http://localhost:1112/`，也能看到 `registered-replicas` 中已经有 peer1 节点，并且这些节点在可用分片（`available-replicase`）之中。我们也可以尝试关闭 peer1，刷新 `http://localhost:1112/`，可以看到 peer1 的节点变为了不可用分片（`unavailable-replicas`）。

- 在设置了多节点的服务注册中心之后，服务提供方还需要做一些简单的配置才能将服务注册到 Eureka Server 集群中。我们以 `hello-service` 为例，修改 `application.properties` 配置文件，如下所示：

```
spring.application.name=hello-service

eureka.client.serviceUrl.defaultZone=http://peer1:1111/eureka/,http://peer2:111
2/eureka/
```

上面的配置主要对 `eureka.client.serviceUrl.defaultZone` 属性做了改动，将注册中心指向了之前我们搭建的 peer1 与 peer2。

下面，我们启动该服务，通过访问 `http://localhost:1111/` 和 `http://localhost:1112/`，可以观察到 `hello-service` 服务同时被注册到了 peer1 和 peer2

上。若此时断开 peer1，由于 `compute-service` 同时也向 peer2 注册，因此在 peer2 上的其他服务依然能访问到 `hello-service`，从而实现了服务注册中心的高可用。

如我们不想使用主机名来定义注册中心的地址，也可以使用 IP 地址的形式，但是需要在配置文件中增加配置参数 `eureka.instance.prefer-ip-address=true`，该值默认为 `false`。

服务发现与消费

通过上面的内容介绍与实践，我们已经搭建起微服务架构中的核心组件——服务注册中心（包括单节点模式和高可用模式）。同时，还对上一章中实现的 Spring Boot 入门程序做了改造。通过简单的配置，使该程序注册到 Eureka 注册中心上，成为该服务治理体系下的一个服务，命名为 `hello-service`。现在我们已经有了服务注册中心和服务提供者，下面就来尝试构建一个服务消费者，它主要完成两个目标，发现服务以及消费服务。其中，服务发现的任务由 Eureka 的客户端完成，而服务消费的任务由 Ribbon 完成。Ribbon 是一个基于 HTTP 和 TCP 的客户端负载均衡器，它可以在通过客户端中配置的 `ribbonServerList` 服务端列表去轮询访问以达到均衡负载的作用。当 Ribbon 与 Eureka 联合使用时，Ribbon 的服务实例清单 `RibbonServerList` 会被 `DiscoveryEnabledNIWSServerList` 重写，扩展成从 Eureka 注册中心中获取服务端列表。同时它也会用 `NIWSDiscoveryPing` 来取代 `IPing`，它将职责委托给 Eureka 来确定服务端是否已经启动。在本章中，我们对 Ribbon 不做详细的介绍，读者只需要理解它在 Eureka 服务发现的基础上，实现了一套对服务实例的选择策略，从而实现对服务的消费。下一章我们会对 Ribbon 做详细的介绍和分析。

下面我们通过构建一个简单的示例，看看在 Eureka 的服务治理体系下如何实现服务的发现与消费。

- 首先，我们做一些准备工作。启动之前实现的服务注册中心 `eureka-server` 以及 `hello-service` 服务，为了实验 Ribbon 的客户端负载均衡功能，我们通过 `java -jar` 命令行的方式来启动两个不同端口的 `hello-service`，具体如下：

```
java -jar hello-service-0.0.1-SNAPSHOT.jar --server.port=8081
java -jar hello-service-0.0.1-SNAPSHOT.jar --server.port=8082
```

- 在成功启动两个 `hello-service` 服务之后，如下图所示，从 Eureka 信息面板中可以看到名为 `HELLO-SERVICE` 的服务中出现了两个实例单元，分别是通过命令行启动的 8081 端口和 8082 端口的服务。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
HELLO-SERVICE	n/a (2)	(2)	UP (2) - PC-201602152056:hello-service:8081 , PC-201602152056:hello-service:8082

- 创建一个 Spring Boot 的基础工程来实现服务消费者，取名为 ribbon-consumer，并在 pom.xml 中引入如下的依赖内容。较之前的 hello-service，我们新增了 Ribbon 模块的依赖 spring-cloud-starter-ribbon。

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.3.7.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</artifactId>
  </dependency>
</dependencies>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Brixton.SR5</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

- 创建应用主类 ConsumerApplication，通过@EnableDiscoveryClient 注解让该应用注册为 Eureka 客户端应用，以获得服务发现的能力。同时，在该主类中创建 RestTemplate 的 Spring Bean 实例，并通过@LoadBalanced 注解开启客户端负载均衡。

```
@EnableDiscoveryClient
@SpringBootApplication
public class ConsumerApplication {

  @Bean
  @LoadBalanced
  RestTemplate restTemplate() {
    return new RestTemplate();
  }
  public static void main(String[] args) {
    SpringApplication.run(ConsumerApplication.class, args);
  }

}
```

- 创建 `ConsumerController` 类并实现 `/ribbon-consumer` 接口。在该接口中，通过在上面创建的 `RestTemplate` 来实现对 HELLO-SERVICE 服务提供的 `/hello` 接口进行调用。可以看到这里访问的地址是服务名 HELLO-SERVICE，而不是一个具体的地址，在服务治理框架中，这是一个非常重要的特性，也符合在本章一开始对服务治理的解释。

```
@RestController
public class ConsumerController {

    @Autowired
    RestTemplate restTemplate;

    @RequestMapping(value = "/ribbon-consumer", method = RequestMethod.GET)
    public String helloConsumer() {
        return restTemplate.getForEntity("http://HELLO-SERVICE/hello",
String.class).getBody();
    }

}
```

- 在 `application.properties` 中配置 Eureka 服务注册中心的位置，需要与之前的 HELLO-SERVICE 一样，不然是发现不了该服务的，同时设置该消费者的端口为 9000，不能与之前启动的应用端口冲突。

```
spring.application.name=ribbon-consumer
server.port=9000

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

- 启动 ribbon-consumer 应用后，我们可以在 Eureka 信息面板中看到，当前除了 HELLO-SERVICE 之外，还多了我们实现的 RIBBON-CONSUMER 服务。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
HELLO-SERVICE	n/a (2)	(2)	UP (2) - PC-201602152056:hello-service:8081 , PC-201602152056:hello-service:8082
RIBBON-CONSUMER	n/a (1)	(1)	UP (1) - PC-201602152056:ribbon-consumer:9000

- 通过向 http://localhost:9000/ribbon-consumer 发起 GET 请求，成功返回了 “Hello World”。此时，我们可以在 ribbon-consumer 应用的控制台中看到如下信息，Ribbon 输出了当前客户端维护的 HELLO-SERVICE 的服务列表情况。其中包含了各个实例的位置，Ribbon 就是按照此信息进行轮询访问，以实现基于客户端的负载均衡。另外还输出了一些其他非常有用的信息，如对各个实例的请求总数量、第一次连接信息、上一次连接信息、总的请求失败数量等。

```
c.n.l.DynamicServerListLoadBalancer      : DynamicServerListLoadBalancer for client
HELLO-SERVICE initialized:
DynamicServerListLoadBalancer:{NFLoadBalancer:name=HELLO-SERVICE,current list of
Servers=[PC-201602152056:8082, PC-201602152056:8081],Load balancer stats=Zone stats:
{defaultzone=[Zone:defaultzone;  Instance count:2; Active connections count: 0;
Circuit breaker tripped count: 0; Active connections per server: 0.0;]
},Server stats: [[Server:PC-201602152056:8082;  Zone:defaultZone; Total Requests:0;
Successive connection failure:0;  Total blackout seconds:0; Last connection made:Thu Jan
01 08:00:00 CST 1970;  First connection made: Thu Jan 01 08:00:00 CST 1970;  Active
Connections:0; total failure count in last (1000) msecs:0; average resp time:0.0;  90
percentile resp time:0.0;  95 percentile resp time:0.0;  min resp time:0.0;  max resp
time:0.0;  stddev resp time:0.0]
, [Server:PC-201602152056:8081; Zone:defaultZone; Total Requests:0; Successive
connection failure:0;  Total blackout seconds:0; Last connection made:Thu Jan 01 08:00:00
CST 1970;  First connection made: Thu Jan 01 08:00:00 CST 1970;  Active Connections:0;
total failure count in last (1000) msecs:0; average resp time:0.0;  90 percentile resp
time:0.0;  95 percentile resp time:0.0;  min resp time:0.0;  max resp time:0.0;  stddev
resp time:0.0]
]}ServerList:org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServer
List@cc7240
```

再尝试发送几次请求，并观察启动的两个 HELLO-SERVICE 的控制台，可以看到两个控制台会交替打印下面的日志，这是我们之前在 HelloController 中实现的对服务信息的输出，可以用来判断当前 ribbon-consumer 对 HELLO-SERVICE 的调用是否是负载均衡的。

```
    com.didispace.web.HelloController        : /hello, host:PC-201602152056,
service_id:hello-service
```

Eureka 详解

在上一节中，我们通过一个简单的服务注册与发现示例，构建了 Eureka 服务治理体系

中的三个核心角色：服务注册中心、服务提供者以及服务消费者。通过上述示例，相信读者对于 Eureka 的服务治理机制已经有了一些初步的认识。至此，我们已经学会了如何构建服务注册中心（包括单节点和高可用部署），也知道了如何使用 Eureka 的注解和配置将 Spring Boot 应用纳入 Eureka 的服务治理体系，成为服务提供者或是服务消费者。同时，对于客户端负载均衡的服务消费也有了一些简单的接触。但是，在实践中，我们的系统结构往往都要比上述示例复杂得多，如果仅仅依靠之前构建的服务治理内容，大多数情况是无法完全直接满足业务系统需求的，我们还需要根据实际情况来做一些配置、调整和扩展。所以，在本节中，我们将详细介绍 Eureka 的基础架构、节点间的通信机制以及一些进阶的配置等。

基础架构

在“服务治理”一节中，我们所讲解的示例虽然简单，但是麻雀虽小、五脏俱全。它已经包含了整个 Eureka 服务治理基础架构的三个核心要素。

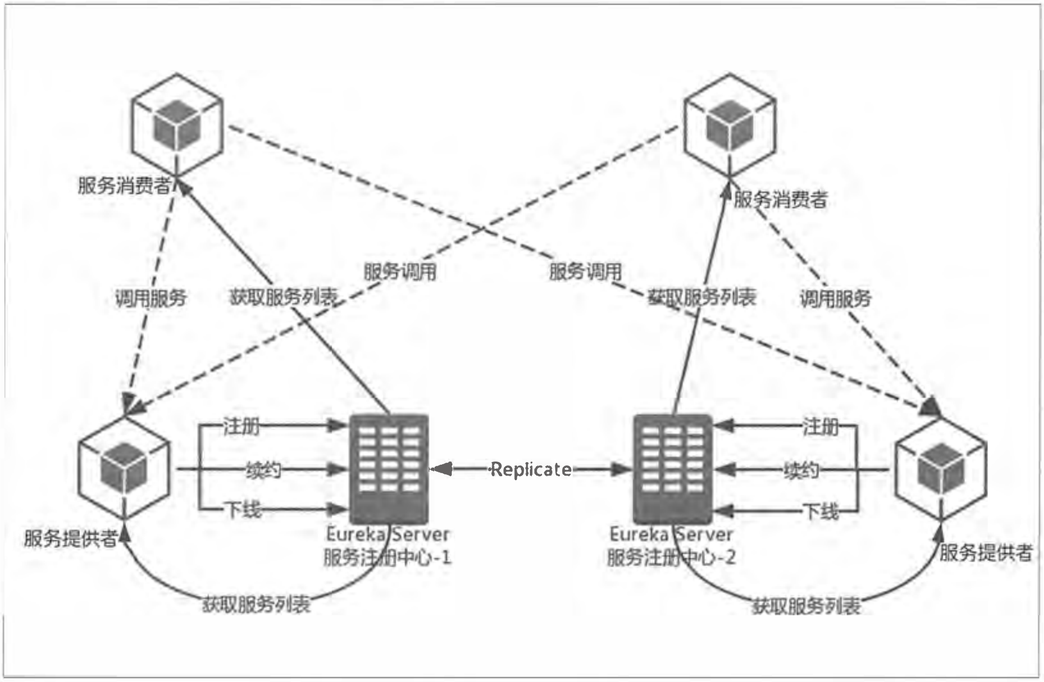
- **服务注册中心**：Eureka 提供的服务端，提供服务注册与发现的功能，也就是在上一节中我们实现的 `eureka-server`。
- **服务提供者**：提供服务的应用，可以是 Spring Boot 应用，也可以是其他技术平台且遵循 Eureka 通信机制的应用。它将自己提供的服务注册到 Eureka，以供其他应用发现，也就是在上一节中我们实现的 `HELLO-SERVICE` 应用。
- **服务消费者**：消费者应用从服务注册中心获取服务列表，从而使消费者可以知道去何处调用其所需要的服务，在上一节中使用了 Ribbon 来实现服务消费，另外后续还会介绍使用 Feign 的消费方式。

很多时候，客户端既是服务提供者也是服务消费者。

服务治理机制

在体验了 Spring Cloud Eureka 通过简单的注解配置就能实现强大的服务治理功能之后，我们来进一步了解一下 Eureka 基础架构中各个元素的一些通信行为，以此来理解基于 Eureka 实现的服务治理体系是如何运作起来的。以下图为例，其中有这样几个重要元素：

- “服务注册中心-1”和“服务注册中心-2”，它们互相注册组成了高可用集群。
- “服务提供者”启动了两个实例，一个注册到“服务注册中心-1”上，另外一个注册到“服务注册中心-2”上。
- 还有两个“服务消费者”，它们也都分别只指向了一个注册中心。



根据上面的结构，下面我们来详细了解一下，从服务注册开始到服务调用，及各个元素所涉及的一些重要通信行为。

服务提供者

服务注册

“服务提供者”在启动的时候会通过发送 REST 请求的方式将自己注册到 Eureka Server 上，同时带上了自身服务的一些元数据信息。Eureka Server 接收到这个 REST 请求之后，将元数据信息存储在一个双层结构 Map 中，其中第一层的 key 是服务名，第二层的 key 是具体服务的实例名。（我们可以回想一下之前在实现 Ribbon 负载均衡的例子中，Eureka 信息面板中一个服务有多个实例的情况，这些内容就是以这样的双层 Map 形式存储的。）

在服务注册时，需要确认一下 `eureka.client.register-with-eureka=true` 参数是否正确，该值默认为 `true`。若设置为 `false` 将不会启动注册操作。

服务同步

如架构图中所示，这里的两个服务提供者分别注册到了两个不同的服务注册中心上，也就是说，它们的信息分别被两个服务注册中心所维护。此时，由于服务注册中心之间因互相注册为服务，当服务提供者发送注册请求到一个服务注册中心时，它会将该请求转发

给集群中相连的其他注册中心，从而实现注册中心之间的服务同步。通过服务同步，两个服务提供者的服务信息就可以通过这两台服务注册中心中的任意一台获取到。

服务续约

在注册完服务之后，服务提供者会维护一个心跳用来持续告诉 Eureka Server："我还活着"，以防止 Eureka Server 的"剔除任务"将该服务实例从服务列表中排除出去，我们称该操作为服务续约（Renew）。

关于服务续约有两个重要属性，我们可以关注并根据需要来进行调整：

```
eureka.instance.lease-renewal-interval-in-seconds=30
eureka.instance.lease-expiration-duration-in-seconds=90
```

`eureka.instance.lease-renewal-interval-in-seconds` 参数用于定义服务续约任务的调用间隔时间，默认为 30 秒。`eureka.instance.lease-expiration-duration-in-seconds` 参数用于定义服务失效的时间，默认为 90 秒。

服务消费者

获取服务

到这里，在服务注册中心已经注册了一个服务，并且该服务有两个实例。当我们启动服务消费者的时候，它会发送一个 REST 请求给服务注册中心，来获取上面注册的服务清单。为了性能考虑，Eureka Server 会维护一份只读的服务清单来返回给客户端，同时该缓存清单会每隔 30 秒更新一次。

获取服务是服务消费者的基础，所以必须确保 `eureka.client.fetch-registry=true` 参数没有被修改成 `false`，该值默认为 `true`。若希望修改缓存清单的更新时间，可以通过 `eureka.client.registry-fetch-interval-seconds=30` 参数进行修改，该参数默认值为 30，单位为秒。

服务调用

服务消费者在获取服务清单后，通过服务名可以获得具体提供服务的实例名和该实例的元数据信息。因为有这些服务实例的详细信息，所以客户端可以根据自己的需要决定具体调用哪个实例，在 Ribbon 中会默认采用轮询的方式进行调用，从而实现客户端的负载均衡。

对于访问实例的选择，Eureka 中有 Region 和 Zone 的概念，一个 Region 中可以包含多个 Zone，每个服务客户端需要被注册到一个 Zone 中，所以每个客户端对应一个 Region 和一个 Zone。在进行服务调用的时候，优先访问同处一个 Zone 中的服务提供方，若访问不到，就访问其他的 Zone，更多关于 Region 和 Zone 的知识，我们会在后续的源码解读中介绍。

服务下线

在系统运行过程中必然会面临关闭或重启服务的某个实例的情况，在服务关闭期间，我们自然不希望客户端会继续调用关闭了的实例。所以在客户端程序中，当服务实例进行正常的关闭操作时，它会触发一个服务下线的 REST 请求给 Eureka Server，告诉服务注册中心：“我要下线了”。服务端在接收到请求之后，将该服务状态置为下线（DOWN），并把该下线事件传播出去。

服务注册中心

失效剔除

有些时候，我们的服务实例并不一定会正常下线，可能由于内存溢出、网络故障等原因使得服务不能正常工作，而服务注册中心并未收到“服务下线”的请求。为了从服务列表中将这些无法提供服务的实例剔除，Eureka Server 在启动的时候会创建一个定时任务，默认每隔一段时间（默认为 60 秒）将当前清单中超时（默认为 90 秒）没有续约的服务剔除出去。

自我保护

当我们在本地调试基于 Eureka 的程序时，基本上都会碰到这样一个问题，在服务注册中心的信息面板中出现类似下面的红色警告信息：

```
    EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT.
RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST
TO BE SAFE.
```

实际上，该警告就是触发了 Eureka Server 的自我保护机制。之前我们介绍过，服务注册到 Eureka Server 之后，会维护一个心跳连接，告诉 Eureka Server 自己还活着。Eureka Server 在运行期间，会统计心跳失败的比例在 15 分钟之内是否低于 85%，如果出现低于的情况（在单机调试的时候很容易满足，实际在生产环境上通常是由于网络不稳定导致），Eureka Server 会将当前的实例注册信息保护起来，让这些实例不会过期，尽可能保护这些注册信息。但是，在这段保护期间内实例若出现问题，那么客户端很容易拿到实际已经不存在的服务实例，会出现调用失败的情况，所以客户端必须要有容错机制，比如可以使用请求重试、断路器等机制。

由于本地调试很容易触发注册中心的保护机制，这会使得注册中心维护的服务实例不那么准确。所以，我们在本地进行开发的时候，可以使用 eureka.server.enable-self-preservation=false 参数来关闭保护机制，以确保注册中心可以将不可用的实例正确剔除。

源码分析

上面，我们对 Eureka 中各个核心元素的通信行为做了详细的介绍，相信大家已经对 Eureka 的运行机制有了一定的了解。为了更深入地理解它的运作和配置，下面我们结合源码来分别看看各个通信行为是如何实现的。

在看具体源码之前，我们先回顾一下之前所实现的内容，从而找到一个合适的切入口去分析。首先，对于服务注册中心、服务提供者、服务消费者这三个主要元素来说，后两者（也就是 Eureka 客户端）在整个运行机制中是大部分通信行为的主动发起者，而注册中心主要是处理请求的接收者。所以，我们可以从 Eureka 的客户端作为入口看看它是如何完成这些主动通信行为的。

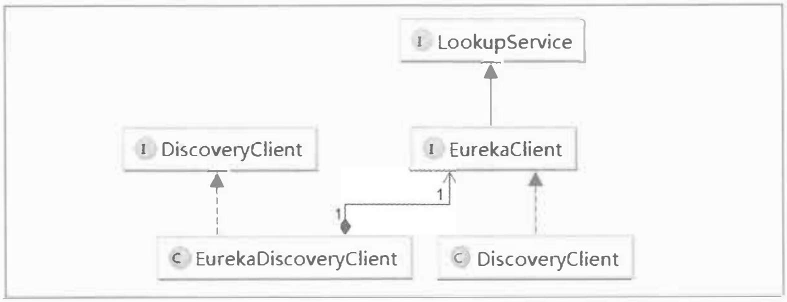
我们在将一个普通的 Spring Boot 应用注册到 Eureka Server 或是从 Eureka Server 中获取服务列表时，主要就做了两件事：

- 在应用主类中配置了`@EnableDiscoveryClient` 注解。
- 在 `application.properties` 中用 `eureka.client.serviceUrl.defaultZone` 参数指定了服务注册中心的位置。

顺着上面的线索，我们来看看`@EnableDiscoveryClient` 的源码，具体如下：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import(EnableDiscoveryClientImportSelector.class)
public @interface EnableDiscoveryClient {

}
```

从该注解的注释中我们可以知道，它主要用来开启 `DiscoveryClient` 的实例。通过搜索 `DiscoveryClient`，我们可以发现有一个类和一个接口。通过梳理可以得到如下图所示的关系：



其中，左边的 org.springframework.cloud.client.discovery.DiscoveryClient 是 Spring Cloud 的接口，它定义了用来发现服务的常用抽象方法，通过该接口可以有效地屏蔽服务治理的实现细节，所以使用 Spring Cloud 构建的微服务应用可以方便地切换不同服务治理框架，而不改动程序代码，只需要另外添加一些针对服务治理框架的配置即可。org.springframework.cloud.netflix.eureka.EurekaDiscoveryClient 是对该接口的实现，从命名来判断，它实现的是对 Eureka 发现服务的封装。所以 EurekaDiscoveryClient 依赖了 Netflix Eureka 的 com.netflix.discovery.EurekaClient 接口，EurekaClient 继承了 LookupService 接口，它们都是 Netflix 开源包中的内容，主要定义了针对 Eureka 的发现服务的抽象方法，而真正实现发现服务的则是 Netflix 包中的 com.netflix.discovery.DiscoveryClient 类。

接下来，我们就来详细看看 DiscoveryClient 类吧。先解读一下该类头部的注释，注释的大致内容如下所示：

```
这个类用于帮助与 Eureka Server 互相协作。

Eureka Client 负责下面的任务：
- 向 Eureka Server 注册服务实例
- 向 Eureka Server 服务租约
- 当服务关闭期间，向 Eureka Server 取消租约
- 查询 Eureka Server 中的服务实例列表

Eureka Client 还需要配置一个 Eureka Server 的 URL 列表。
```

在具体研究 Eureka Client 负责完成的任务之前，我们先看看在哪里对 Eureka Server 的 URL 列表进行配置。根据我们配置的属性名 eureka.client.serviceUrl.defaultZone，通过 serviceUrl 可以找到该属性相关的加载属性，但是在 SR5 版本中它们都被 @Deprecated 标注为不再建议使用，并@link 到了替代类 com.netflix.discovery.endpoint.EndpointUtils，所以我们可以在该类中找到下面这个函数：

```
    public static Map<String, List<String>> getServiceUrlsMapFromConfig(
            EurekaClientConfig clientConfig, String instanceZone, boolean
preferSameZone) {
        Map<String, List<String>> orderedUrls = new LinkedHashMap<>();
        String region = getRegion(clientConfig);
        String[] availZones = clientConfig.getAvailabilityZones(clientConfig.getRegion());
        if (availZones == null || availZones.length == 0) {
            availZones = new String[1];
            availZones[0] = DEFAULT_ZONE;
        }
        ......
```

```
    int myZoneOffset = getZoneOffset(instanceZone, preferSameZone, availZones);

    String zone = availZones[myZoneOffset];
    List<String> serviceUrls = clientConfig.getEurekaServerServiceUrls(zone);
    if (serviceUrls != null) {
        orderedUrls.put(zone, serviceUrls);
    }
    ……
    return orderedUrls;
}
```

Region、Zone

在上面的函数中，可以发现，客户端依次加载了两个内容，第一个是 Region，第二个是 Zone，从其加载逻辑上我们可以判断它们之间的关系：

- 通过 `getRegion` 函数，我们可以看到它从配置中读取了一个 Region 返回，所以一个微服务应用只可以属于一个 Region，如果不特别配置，默认为 default。若我们要自己设置，可以通过 `eureka.client.region` 属性来定义。

```
public static String getRegion(EurekaClientConfig clientConfig) {
    String region = clientConfig.getRegion();
    if (region == null) {
        region = DEFAULT_REGION;
    }
    region = region.trim().toLowerCase();
    return region;
}
```

- 通过 `getAvailabilityZones` 函数，可以知道当我们没有特别为 Region 配置 Zone 的时候，将默认采用 defaultZone，这也是我们之前配置参数 `eureka.client.serviceUrl.defaultZone` 的由来。若要为应用指定 Zone，可以通过 `eureka.client.availability-zones` 属性来进行设置。从该函数的 `return` 内容，我们可以知道 Zone 能够设置多个，并且通过逗号分隔来配置。由此，我们可以判断 Region 与 Zone 是一对多的关系。

```
public String[] getAvailabilityZones(String region) {
    String value = this.availabilityZones.get(region);
    if (value == null) {
        value = DEFAULT_ZONE;
    }
    return value.split(",");
}
```

serviceUrls

在获取了 Region 和 Zone 的信息之后，才开始真正加载 Eureka Server 的具体地址。它根据传入的参数按一定算法确定加载位于哪一个 Zone 配置的 serviceUrls。

```
int myZoneOffset = getZoneOffset(instanceZone, preferSameZone, availZones);
String zone = availZones[myZoneOffset];
List<String> serviceUrls = clientConfig.getEurekaServerServiceUrls(zone);
```

具体获取 serviceUrls 的实现，我们可以详细查看 getEurekaServerServiceUrls 函数的具体实现类 EurekaClientConfigBean，该类是 EurekaClientConfig 和 EurekaConstants 接口的实现，用来加载配置文件中的内容，这里有非常多有用的信息，我们先说一下此处我们关心的，关于 defaultZone 的信息。通过搜索 defaultZone，我们可以很容易找到下面这个函数，它具体实现了如何解析该参数的过程，通过此内容，我们就可以知道，eureka.client.serviceUrl.defaultZone 属性可以配置多个，并且需要通过逗号分隔。

```
public List<String> getEurekaServerServiceUrls(String myZone) {
    String serviceUrls = this.serviceUrl.get(myZone);
    if (serviceUrls == null || serviceUrls.isEmpty()) {
        serviceUrls = this.serviceUrl.get(DEFAULT_ZONE);
    }
    if (!StringUtils.isEmpty(serviceUrls)) {
        final String[] serviceUrlsSplit =
StringUtils.commaDelimitedListToStringArray(serviceUrls);
        List<String> eurekaServiceUrls = new ArrayList<>(serviceUrlsSplit.length);
        for (String eurekaServiceUrl : serviceUrlsSplit) {
            if (!endsWithSlash(eurekaServiceUrl)) {
                eurekaServiceUrl += "/";
            }
            eurekaServiceUrls.add(eurekaServiceUrl);
        }
        return eurekaServiceUrls;
    }
    return new ArrayList<>();
}
```

当我们在微服务应用中使用 Ribbon 来实现服务调用时，对于 Zone 的设置可以在负载均衡时实现区域亲和特性：Ribbon 的默认策略会优先访问同客户端处于一个 Zone 中的服务端实例，只有当同一个 Zone 中没有可用服务端实例的时候才会访问其他 Zone 中的实例。所以通过 Zone 属性的定义，配合实际部署的物理结构，我们就可以有效地设计出对区域性故障的容错集群。

服务注册

在理解了多个服务注册中心信息的加载后，我们再回头看看 DiscoveryClient 类是如何实现“服务注册”行为的，通过查看它的构造类，可以找到它调用了下面这个函数：

```
private void initScheduledTasks() {
    ...
    if (clientConfig.shouldRegisterWithEureka()) {
        ...
        // InstanceInfo replicator
        instanceInfoReplicator = new InstanceInfoReplicator(
                this,
               instanceInfo,
                clientConfig.getInstanceInfoReplicationIntervalSeconds(),
                2); // burstSize
        ...

instanceInfoReplicator.start(clientConfig.getInitialInstanceInfoReplicationInterval
Seconds());
    } else {
        logger.info("Not registering with Eureka server per configuration");
    }
}
```

从上面的函数中，可以看到一个与服务注册相关的判断语句 if (clientConfig.shouldRegisterWithEureka())。在该分支内，创建了一个 InstanceInfoReplicator 类的实例，它会执行一个定时任务，而这个定时任务的具体工作可以查看该类的 run()函数，具体如下所示：

```
public void run() {
    try {
        discoveryClient.refreshInstanceInfo();
        Long dirtyTimestamp = instanceInfo.isDirtyWithTime();
        if (dirtyTimestamp != null) {
            discoveryClient.register();
            instanceInfo.unsetIsDirty(dirtyTimestamp);
        }
    } catch (Throwable t) {
        logger.warn("There was a problem with the instance info replicator", t);
    } finally {
        Future next = scheduler.schedule(this, replicationIntervalSeconds,
TimeUnit.SECONDS);
        scheduledPeriodicRef.set(next);
    }
}
```

相信大家都发现了 discoveryClient.register();这一行，真正触发调用注册的地方就在这里。继续查看 register()的实现内容，如下所示：

```
    boolean register() throws Throwable {
        logger.info(PREFIX + appPathIdentifier + ": registering service...");
        EurekaHttpResponse<Void> httpResponse;
        try {
            httpResponse = eurekaTransport.registrationClient.register(instanceInfo);
        } catch (Exception e) {
            logger.warn("{} - registration failed {}", PREFIX + appPathIdentifier,
e.getMessage(), e);
            throw e;
        }
        if (logger.isInfoEnabled()) {
            logger.info("{} - registration status: {}", PREFIX + appPathIdentifier,
httpResponse.getStatusCode());
        }
        return httpResponse.getStatusCode() == 204;
    }
```

通过属性命名，大家基本也能猜出来，注册操作也是通过 REST 请求的方式进行的。同时，我们能看到发起注册请求的时候，传入了一个 com.netflix.appinfo.InstanceInfo 对象，该对象就是注册时客户端给服务端的服务的元数据。

服务获取与服务续约

顺着上面的思路，我们继续来看 DiscoveryClient 的 initScheduledTasks 函数，不难发现在其中还有两个定时任务，分别是“服务获取”和“服务续约”：

```
    private void initScheduledTasks() {
        if (clientConfig.shouldFetchRegistry()) {
            // registry cache refresh timer
            int registryFetchIntervalSeconds = clientConfig.getRegistryFetchInterval-
Seconds();
            int expBackOffBound = clientConfig.getCacheRefreshExecutorExponential-
BackOffBound();
            scheduler.schedule(
                    new TimedSupervisorTask(
                            "cacheRefresh",
                            scheduler,
                            cacheRefreshExecutor,
                            registryFetchIntervalSeconds,
                            TimeUnit.SECONDS,
                            expBackOffBound,
                            new CacheRefreshThread()
                    ),
                    registryFetchIntervalSeconds, TimeUnit.SECONDS);
```

```
        }
        if (clientConfig.shouldRegisterWithEureka()) {
            int renewalIntervalInSecs = instanceInfo.getLeaseInfo().getRenewalIntervalInSecs();
            int expBackOffBound = clientConfig.getHeartbeatExecutorExponentialBackOffBound();
            logger.info("Starting heartbeat executor: " + "renew interval is: " +
renewalIntervalInSecs);

            // Heartbeat timer
            scheduler.schedule(
                    new TimedSupervisorTask(
                            "heartbeat",
                            scheduler,
                            heartbeatExecutor,
                            renewalIntervalInSecs,
                            TimeUnit.SECONDS,
                            expBackOffBound,
                            new HeartbeatThread()
                    ),
                    renewalIntervalInSecs, TimeUnit.SECONDS);
            // InstanceInfo replicator
            ……
        }
    }
```

从源码中我们可以发现，“服务获取”任务相对于“服务续约”和“服务注册”任务更为独立。“服务续约”与“服务注册”在同一个 if 逻辑中，这个不难理解，服务注册到 Eureka Server 后，自然需要一个心跳去续约，防止被剔除，所以它们肯定是成对出现的。从源码中，我们更清楚地看到了之前所提到的，对于服务续约相关的时间控制参数：

```
eureka.instance.lease-renewal-interval-in-seconds=30
eureka.instance.lease-expiration-duration-in-seconds=90
```

而“服务获取”的逻辑在独立的一个 if 判断中，其判断依据就是我们之前所提到的 `eureka.client.fetch-registry=true` 参数，它默认为 `true`，大部分情况下我们不需要关心。为了定期更新客户端的服务清单，以保证客户端能够访问确实健康的服务实例，“服务获取”的请求不会只限于服务启动，而是一个定时执行的任务，从源码中我们可以看到任务运行中的 `registryFetchIntervalSeconds` 参数对应的就是之前所提到的 `eureka.client.registry-fetch-interval-seconds=30` 配置参数，它默认为 30 秒。

继续向下深入，我们能分别发现实现“服务获取”和“服务续约”的具体方法，其中“服务续约”的实现较为简单，直接以 REST 请求的方式进行续约：

```
boolean renew() {
    EurekaHttpResponse<InstanceInfo> httpResponse;
```

```
        try {
            httpResponse = eurekaTransport.registrationClient.sendHeartBeat
(instanceInfo.getAppName(), instanceInfo.getId(), instanceInfo, null);
            logger.debug("{} - Heartbeat status: {}", PREFIX + appPathIdentifier,
httpResponse.getStatusCode());
            if (httpResponse.getStatusCode() == 404) {
                REREGISTER_COUNTER.increment();
                logger.info("{} - Re-registering apps/{}", PREFIX + appPathIdentifier,
instanceInfo.getAppName());
                return register();
            }
            return httpResponse.getStatusCode() == 200;
        } catch (Throwable e) {
            logger.error("{} - was unable to send heartbeat!", PREFIX + appPathIdentifier, e);
            return false;
        }
    }
```

而“服务获取”则复杂一些，会根据是否是第一次获取发起不同的 REST 请求和相应的处理。具体的实现逻辑跟之前类似，有兴趣的读者可以继续查看服务客户端的其他具体内容，以了解更多细节。

服务注册中心处理

通过上面的源码分析，可以看到所有的交互都是通过 REST 请求来发起的。下面我们来看看服务注册中心对这些请求的处理。Eureka Server 对于各类 REST 请求的定义都位于 com.netflix.eureka.resources 包下。

以“服务注册”请求为例：

```
    @POST
    @Consumes({"application/json", "application/xml"})
    public Response addInstance(InstanceInfo info,
                    @HeaderParam(PeerEurekaNode.HEADER_REPLICATION) String
isReplication) {
        logger.debug("Registering instance {} (replication={})", info.getId(),
isReplication);
        // validate that the instanceinfo contains all the necessary required fields
        ...
        // handle cases where clients may be registering with bad DataCenterInfo with
missing data
        DataCenterInfo dataCenterInfo = info.getDataCenterInfo();
        if (dataCenterInfo instanceof UniqueIdentifier) {
            String dataCenterInfoId = ((UniqueIdentifier) dataCenterInfo).getId();
            if (isBlank(dataCenterInfoId)) {
                boolean experimental = "true".equalsIgnoreCase(
                        serverConfig.getExperimental("registration.validation.
```

```
dataCenterInfoId"));
            if (experimental) {
                String entity = "DataCenterInfo of type " + dataCenterInfo.getClass()
                                        + " must contain a valid id";
                return Response.status(400).entity(entity).build();
            } else if (dataCenterInfo instanceof AmazonInfo) {
                AmazonInfo amazonInfo = (AmazonInfo) dataCenterInfo;
                String effectiveId = amazonInfo.get(AmazonInfo.MetaDataKey.instanceId);
                if (effectiveId == null) {
                    amazonInfo.getMetadata().put(
                            AmazonInfo.MetaDataKey.instanceId.getName(), info.getId());
                }
            } else {
                logger.warn("Registering DataCenterInfo of type {} without an
appropriate id",
                        dataCenterInfo.getClass());
            }
        }
    }

    registry.register(info, "true".equals(isReplication));
    return Response.status(204).build();  // 204 to be backwards compatible
}
```

在对注册信息进行了一堆校验之后，会调用 org.springframework.cloud.netflix.eureka.server.InstanceRegistry 对象中的 register(InstanceInfo info, int leaseDuration, boolean isReplication)函数来进行服务注册：

```
    public void register(InstanceInfo info, int leaseDuration, boolean isReplication) {
        if (log.isDebugEnabled()) {
            log.debug("register " + info.getAppName() + ", vip " + info.getVIPAddress()
                    + ", leaseDuration " + leaseDuration + ", isReplication "
                    + isReplication);
        }
        this.ctxt.publishEvent(new EurekaInstanceRegisteredEvent(this, info,
                leaseDuration, isReplication));

        super.register(info, leaseDuration, isReplication);
    }
```

在注册函数中，先调用 publishEvent 函数，将该新服务注册的事件传播出去，然后调用 com.netflix.eureka.registry.AbstractInstanceRegistry 父类中的注册实现，将 InstanceInfo 中的元数据信息存储在一个 ConcurrentHashMap 对象中。正如我们之前所说的，注册中心存储了两层 Map 结构，第一层的 key 存储服务名：InstanceInfo 中的 appName 属性，第二层的 key 存储实例名：InstanceInfo 中的

instanceId 属性。

服务端的请求和接收非常类似，对于其他的服务端处理，这里不再展开叙述，读者可以根据上面的脉络来自己查看其内容（这里包含很多细节内容）来帮助和加深理解。

配置详解

在分析了 Eureka 的部分源码之后，相信大家对 Eureka 的服务治理机制已经有了进一步的理解。在本节中，我们从使用的角度对 Eureka 中一些常用配置内容进行详细的介绍，以帮助我们根据自身环境与业务特点来进行个性化的配置调整。

在 Eureka 的服务治理体系中，主要分为服务端与客户端两个不同的角色，服务端为服务注册中心，而客户端为各个提供接口的微服务应用。当我们构建了高可用的注册中心之后，该集群中所有的微服务应用和后续将要介绍的一些基础类应用（如配置中心、API 网关等）都可以视作该体系下的一个微服务（Eureka 客户端）。服务注册中心也一样，只是高可用环境下的服务注册中心除了作为客户端之外，还为集群中的其他客户端提供了服务注册的特殊功能。所以，Eureka 客户端的配置对象存在于所有 Eureka 服务治理体系下的应用实例中。在实际使用 Spring Cloud Eureka 的过程中，我们所做的配置内容几乎都是对 Eureka 客户端配置进行的操作，所以了解这部分的配置内容，对于用好 Eureka 非常有帮助。

Eureka 客户端的配置主要分为以下两个方面。

- 服务注册相关的配置信息，包括服务注册中心的地址、服务获取的间隔时间、可用区域等。
- 服务实例相关的配置信息，包括服务实例的名称、IP 地址、端口号、健康检查路径等。

而 Eureka 服务端更多地类似于一个现成产品，大多数情况下，我们不需要修改它的配置信息。所以在本书中，我们对此不进行过多的介绍，有兴趣的读者可以查看 org.springframework.cloud.netflix.eureka.server.EurekaServerConfig-Bean 类的定义来做进一步的学习，这些参数均以 eureka.server 作为前缀。另外值得一提的是，我们在学习本书内容进行本地调试的时候，可以通过设置该类中的 enableSelfPreservation 参数来关闭注册中心的“自我保护”功能，以防止关闭的实例无法被服务注册中心剔除的问题，这一点我们在“服务治理机制”一节中也有所介绍。

服务注册类配置

关于服务注册类的配置信息，我们可以通过查看 org.springframework.cloud.netflix.eureka.EurekaClientConfigBean 的源码来获得比官方文档中更为详尽

的内容，这些配置信息都以 eureka.client 为前缀。下面我们针对一些常用的配置信息做进一步的介绍和说明。

指定注册中心

在本章第 1 节的示例中，我们演示了如何将一个 Spring Boot 应用纳入 Eureka 的服务治理体系，除了引入 Eureka 的依赖之外，就是在配置文件中指定注册中心，主要通过 eureka.client.serviceUrl 参数实现。该参数的定义如下所示，它的配置值存储在 HashMap 类型中，并且设置有一组默认值，默认值的 key 为 defaultZone、value 为 http://localhost:8761/eureka/。

```
private Map<String, String> serviceUrl = new HashMap<>();
{
    this.serviceUrl.put(DEFAULT_ZONE, DEFAULT_URL);
}

public static final String DEFAULT_URL = "http://localhost:8761" + DEFAULT_PREFIX
+ "/";
public static final String DEFAULT_ZONE = "defaultZone";
```

由于之前实现的服务注册中心使用了 1111 端口，所以我们做了如下配置，来将应用注册到对应的 Eureka 服务端中。

```
eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

当构建了高可用的服务注册中心集群时，我们可以为参数的 value 值配置多个注册中心的地址（通过逗号分隔）。比如下面的例子：

```
eureka.client.serviceUrl.defaultZone=http://peer1:1111/eureka/,http://peer2:111
2/eureka/
```

另外，为了服务注册中心的安全考虑，很多时候我们都会为服务注册中心加入安全校验。这个时候，在配置 serviceUrl 时，需要在 value 值的 URL 中加入相应的安全校验信息，比如 http://<username>:<password>@localhost:1111/eureka。其中，<username>为安全校验信息的用户名，<password>为该用户的密码。

其他配置

下面整理了 org.springframework.cloud.netflix.eureka.EurekaClientConfigBean 中定义的常用配置参数以及对应的说明和默认值，这些参数均以 eureka.client 为前缀。

参数名	说明	默认值
enabled	启用 Eureka 客户端	true

续表

参数名	说明	默认值
registryFetchIntervalSeconds	从 Eureka 服务端获取注册信息的间隔时间，单位为秒	30
instanceInfoReplicationIntervalSeconds	更新实例信息的变化到 Eureka 服务端的间隔时间，单位为秒	30
initialInstanceInfoReplicationIntervalSeconds	初始化实例信息到 Eureka 服务端的间隔时间，单位为秒	40
eurekaServiceUrlPollIntervalSeconds	轮询 Eureka 服务端地址更改的间隔时间，单位为秒。当我们与 Spring Cloud Config 配合，动态刷新 Eureka 的 serviceURL 地址时需要关注该参数	300
eurekaServerReadTimeoutSeconds	读取 Eureka Server 信息的超时时间，单位为秒	8
eurekaServerConnectTimeoutSeconds	连接 Eureka Server 的超时时间，单位为秒	5
eurekaServerTotalConnections	从 Eureka 客户端到所有 Eureka 服务端的连接总数	200
eurekaServerTotalConnectionsPerHost	从 Eureka 客户端到每个 Eureka 服务端主机的连接总数	50
eurekaConnectionIdleTimeoutSeconds	Eureka 服务端连接的空闲关闭时间，单位为秒	30
heartbeatExecutorThreadPoolSize	心跳连接池的初始化线程数	2
heartbeatExecutorExponentialBackOffBound	心跳超时重试延迟时间的最大乘数值	10
cacheRefreshExecutorThreadPoolSize	缓存刷新线程池的初始化线程数	2
cacheRefreshExecutorExponentialBackOffBound	缓存刷新重试延迟时间的最大乘数值	10
useDnsForFetchingServiceUrls	使用 DNS 来获取 Eureka 服务端的 serviceUrl	false
registerWithEureka	是否要将自身的实例信息注册到Eureka服务端	true
preferSameZoneEureka	是否偏好使用处于相同 Zone 的 Eureka 服务端	true
filterOnlyUpInstances	获取实例时是否过滤，仅保留 UP 状态的实例	true
fetchRegistry	是否从 Eureka 服务端获取注册信息	true

服务实例类配置

关于服务实例类的配置信息，我们可以通过查看 `org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean` 的源码来获取详细内容，这些配置信息都以 `eureka.instance` 为前缀，下面我们针对一些常用的配置信息做一些详细的说明。

元数据

在 org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean 的配置信息中，有一大部分内容都是对服务实例元数据的配置，那么什么是服务实例的元数据呢？它是 Eureka 客户端在向服务注册中心发送注册请求时，用来描述自身服务信息的对象，其中包含了一些标准化的元数据，比如服务名称、实例名称、实例 IP、实例端口等用于服务治理的重要信息；以及一些用于负载均衡策略或是其他特殊用途的自定义元数据信息。

在使用 Spring Cloud Eureka 的时候，所有的配置信息都通过 org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean 进行加载，但在真正进行服务注册的时候，还是会包装成 com.netflix.appinfo.InstanceInfo 对象发送给 Eureka 服务端。这两个类的定义非常相似，我们可以直接查看 com.netflix.appinfo.InstanceInfo 类中的详细定义来了解原生 Eureka 对元数据的定义。其中，Map<String, String> metadata = new ConcurrentHashMap<String, String>() 是自定义的元数据信息，而其他成员变量则是标准化的元数据信息。Spring Cloud 的 EurekaInstanceConfigBean 对原生元数据对象做了一些配置优化处理，在后续的介绍中，我们也会提到这些内容。

我们可以通过 eureka.instance.<properties>=<value>的格式对标准化元数据直接进行配置，其中<properties>就是 EurekaInstanceConfigBean 对象中的成员变量名。而对于自定义元数据，可以通过 eureka.instance.metadataMap.<key>=<value>的格式来进行配置，比如：

```
eureka.instance.metadataMap.zone=shanghai
```

接下来，我们将针对一些常用的元数据配置做进一步的介绍和说明。

实例名配置

实例名，即 InstanceInfo 中的 instanceId 参数，它是区分同一服务中不同实例的唯一标识。在 Netflix Eureka 的原生实现中，实例名采用主机名作为默认值，这样的设置使得在同一主机上无法启动多个相同的服务实例。所以，在 Spring Cloud Eureka 的配置中，针对同一主机中启动多实例的情况，对实例名的默认命名做了更为合理的扩展，它采用了如下默认规则：

```
${spring.cloud.client.hostname}:${spring.application.name}:${spring.application
.instance_id:${server.port}}
```

对于实例名的命名规则，我们可以通过 eureka.instance.instanceId 参数来进行配置。比如，在本地进行客户端负载均衡调试时，需要启动同一服务的多个实例，如果我们直接启动同一个应用必然会产生端口冲突。虽然可以在命令行中指定不同的

server.port 来启动，但是这样还是略显麻烦。实际上，我们可以直接通过设置 server.port=0 或者使用随机数 server.port=${random.int[10000,19999]}来让 Tomcat 启动的时候采用随机端口。但是这个时候我们会发现注册到 Eureka Server 的实例名都是相同的，这会使得只有一个服务实例能够正常提供服务。对于这个问题，我们就可以通过设置实例名规则来轻松解决：

```
eureka.instance.instanceId=${spring.application.name}:${random.int}}
```

通过上面的配置，利用应用名加随机数的方式来区分不同的实例，从而实现在同一主机上，不指定端口就能轻松启动多个实例的效果。

端点配置

在 InstanceInfo 中，我们可以看到一些 URL 的配置信息，比如 homePageUrl、statusPageUrl、healthCheckUrl，它们分别代表了应用主页的 URL、状态页的 URL、健康检查的 URL。其中，状态页和健康检查的 URL 在 Spring Cloud Eureka 中默认使用了 spring-boot-actuator 模块提供的/info 端点和/health 端点。虽然我们在之前的示例中并没有对这些端点做具体的设置，但是实际上这些 URL 地址的配置非常重要。为了服务的正常运作，我们必须确保 Eureka 客户端的/health 端点在发送元数据的时候，是一个能够被注册中心访问到的地址，否则服务注册中心不会根据应用的健康检查来更改状态（仅当开启了 healthcheck 功能时，以该端点信息作为健康检查标准）。而/info 端点如果不正确的话，会导致在 Eureka 面板中单击服务实例时，无法访问到服务实例提供的信息接口。

大多数情况下，我们并不需要修改这几个 URL 的配置，但是在一些特殊情况下，比如，为应用设置了 context-path，这时，所有 spring-boot-actuator 模块的监控端点都会增加一个前缀。所以，我们就需要做类似如下的配置，为/info 和/health 端点也加上类似的前缀信息：

```
management.context-path=/hello

eureka.instance.statusPageUrlPath=${management.context-path}/info
eureka.instance.healthCheckUrlPath=${management.context-path}/health
```

另外，有时候为了安全考虑，也有可能会修改/info 和/health 端点的原始路径。这个时候，我们也需要做一些特殊的配置，比如像下面这样：

```
endpoints.info.path=/appInfo
endpoints.health.path=/checkHealth

eureka.instance.statusPageUrlPath=/${endpoints.info.path}
eureka.instance.healthCheckUrlPath=/${endpoints.health.path}
```

在上面所举的两个示例中，我们使用了 eureka.instance.statusPageUrlPath 和 eureka.instance.healthCheckUrlPath 参数，这两个配置值有一个共同特点，它们都使用相对路径来进行配置。由于 Eureka 的服务注册中心默认会以 HTTP 的方式来访问和暴露这些端点，因此当客户端应用以 HTTPS 的方式来暴露服务和监控端点时，相对路径的配置方式就无法满足需求了。所以，Spring Cloud Eureka 还提供了绝对路径的配置参数，具体示例如下所示：

```
eureka.instance.statusPageUrl=https://${eureka.instance.hostname}/info
eureka.instance.healthCheckUrl=https://${eureka.instance.hostname}/health
eureka.instance.homePageUrl=https://${eureka.instance.hostname}/
```

健康检测

默认情况下，Eureka 中各个服务实例的健康检测并不是通过 spring-boot-actuator 模块的/health 端点来实现的，而是依靠客户端心跳的方式来保持服务实例的存活。在 Eureka 的服务续约与剔除机制下，客户端的健康状态从注册到注册中心开始都会处于 UP 状态，除非心跳终止一段时间之后，服务注册中心将其剔除。默认的心跳实现方式可以有效检查客户端进程是否正常运作，但却无法保证客户端应用能够正常提供服务。由于大多数微服务应用都会有一些其他的外部资源依赖，比如数据库、缓存、消息代理等，如果我们的应用与这些外部资源无法联通的时候，实际上已经不能提供正常的对外服务了，但是因为客户端心跳依然在运行，所以它还是会被服务消费者调用，而这样的调用实际上并不能获得预期的结果。

在 Spring Cloud Eureka 中，我们可以通过简单的配置，把 Eureka 客户端的健康检测交给 spring-boot-actuator 模块的/health 端点，以实现更加全面的健康状态维护。详细的配置步骤如下所示：

- 在 pom.xml 中引入 spring-boot-starter-actuator 模块的依赖。
- 在 application.properties 中增加参数配置 eureka.client.healthcheck.enabled=true。
- 如果客户端的/health 端点路径做了一些特殊处理，请参考前文介绍端点配置时的方法进行配置，让服务注册中心可以正确访问到健康检测端点。

其他配置

除了上面介绍的配置参数外，下面整理了一些 org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean 中定义的配置参数以及对应的说明和默认值，这些参数均以 eureka.instance 为前缀。

参数名	说明	默认值
preferIpAddress	是否优先使用 IP 地址作为主机名的标识	false
leaseRenewalIntervalInSeconds	Eureka 客户端向服务端发送心跳的时间间隔，单位为秒	30
leaseExpirationDurationInSeconds	Eureka 服务端在收到最后一次心跳之后等待的时间上限，单位为秒。超过该时间之后服务端会将该服务实例从服务清单中剔除，从而禁止服务调用请求被发送到该实例上	90
nonSecurePort	非安全的通信端口号	80
securePort	安全的通信端口号	443
nonSecurePortEnabled	是否启用非安全的通信端口号	true
securePortEnabled	是否启用安全的通信端口号	
appname	服务名，默认取 `spring.application.name` 的配置值，如果没有则为 `unknown`	
hostname	主机名，不配置的时候将根据操作系统的主机名来获取	

在上面的这些配置中，除了前三个配置参数在需要的时候可以做一些调整，其他的参数配置大多数情况下不需要进行配置，使用默认值即可。

跨平台支持

在“Eureka 详解”一节中，我们对 Spring Cloud Eureka 的源码做了较为详细的分析，在分析过程中相信大家已经发现，Eureka 的通信机制使用了 HTTP 的 REST 接口实现，这也是 Eureka 同其他服务注册工具的一个关键不同点。由于 HTTP 的平台无关性，虽然 Eureka Server 通过 Java 实现，但是在其下的微服务应用并不限于使用 Java 来进行开发。

跨平台本身就是微服务架构的九大特性之一，只有实现了对技术平台的透明，才能更好地发挥不同语言对不同业务处理能力的优势，从而打造更为强大的大型系统。

目前除了 Eureka 的 Java 客户端之外，还有很多其他语言平台对其的支持，比如 eureka-js-client、python-eureka 等。若我们有志于自己为一门语言来开发客户端程序，也并非特别复杂，只需要根据上面提到的那些用户服务协调的通信请求实现就能实现服务的注册与发现，需要了解更多关于 Eureka 的 REST API 内容可以查看 Eureka 官方 WiKi 中的 *Eureka-REST-operations* 一文 https://github.com/Netflix/eureka/wiki/Eureka-REST-operations。

通信协议

默认情况下，Eureka 使用 Jersey 和 XStream 配合 JSON 作为 Server 与 Client 之间的通信协议。你也可以选择实现自己的协议来代替。

- Jersey 是 JAX-RS 的参考实现，它包含三个主要部分。
  - **核心服务器**（Core Server）：通过提供 JSR 311 中标准化的注释和 API 标准化，你可以用直观的方式开发 RESTf ulWeb 服务。
  - **核心客户端**（Core Client）：Jersey 客户端 API 帮助你与 REST 服务轻松通信。
  - **集成**（Integration）：Jersey 还提供可以轻松集成 Spring、Guice、Apache Abdera 的库。
- XStream 是用来将对象序列化成 XML（JSON）或反序列化为对象的一个 Java 类库。XStream 在运行时使用 Java 反射机制对要进行序列化的对象树的结构进行探索，并不需要对对象做出修改。XStream 可以序列化内部字段，包括 private 和 f inal 字段，并且支持非公开类以及内部类。默认情况下，XStream 不需要配置映射关系，对象和字段将映射为同名 XML 元素。但是当对象和字段名与 XML 中的元素名不同时，XStream 支持指定别名。XStream 支持以方法调用的方式，或是 Java 标注的方式指定别名。XStream 在进行数据类型转换时，使用系统默认的类型转换器。同时，也支持用户自定义的类型转换器。

JAX-RS 是将在 Java EE 6 中引入的一种新技术。JAX-RS 即 Java API for RESTful Web Services，是一个 Java 编程语言的应用程序接口，支持按照表述性状态转移（REST）架构风格创建 Web 服务。JAX-RS 使用了 Java SE 5 引入的 Java 标注来简化 Web 服务的客户端和服务端的开发和部署。包括：

- @Path，标注资源类或者方法的相对路径。
- @GET、@PUT、@POST、@DELETE，标注方法是 HTTP 请求的类型。
- @Produces，标注返回的 MIME 媒体类型。
- @Consumes，标注可接受请求的 MIME 媒体类型。
- @PathParam、@QueryParam、@HeaderParam、@CookieParam、@MatrixParam、@FormParam，标注方法的参数来自 HTTP 请求的不同位置，例如，@PathParam 来自 URL 的路径，@QueryParam 来自 URL 的查询参数，@HeaderParam 来自 HTTP 请求的头信息，@CookieParam 来自 HTTP 请求的 Cookie。

之前在分析 Eureka Server 端源码的时候，查看请求处理时可以看到很多上面这些注解的定义。

第 4 章

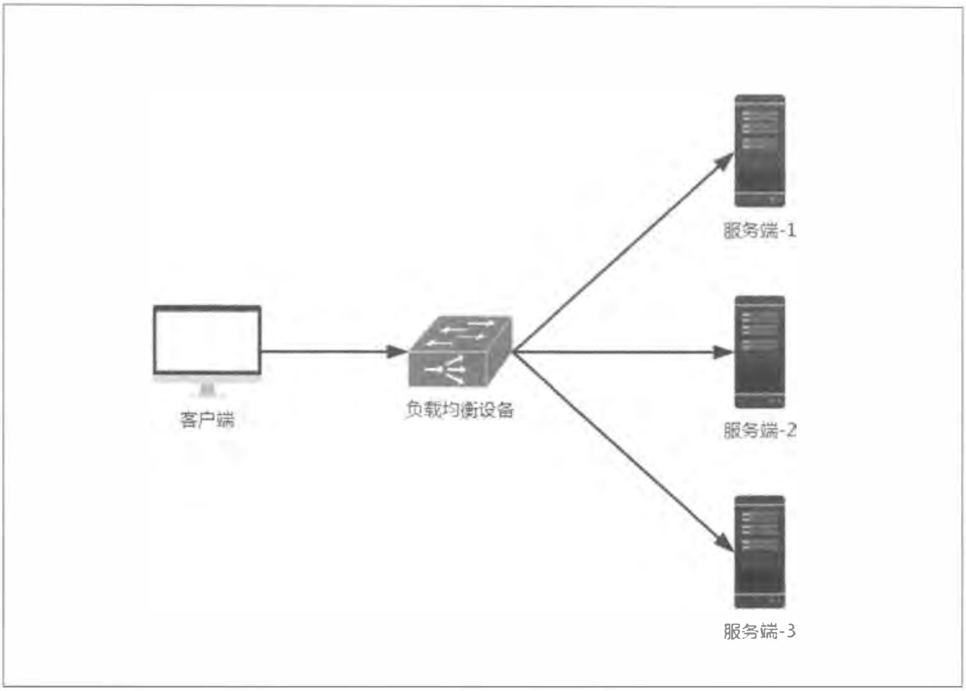
客户端负载均衡：Spring Cloud Ribbon

Spring Cloud Ribbon 是一个基于 HTTP 和 TCP 的客户端负载均衡工具，它基于 Netflix Ribbon 实现。通过 Spring Cloud 的封装，可以让我们轻松地将面向服务的 REST 模板请求自动转换成客户端负载均衡的服务调用。Spring Cloud Ribbon 虽然只是一个工具类框架，它不像服务注册中心、配置中心、API 网关那样需要独立部署，但是它几乎存在于每一个 Spring Cloud 构建的微服务和基础设施中。因为微服务间的调用，API 网关的请求转发等内容，实际上都是通过 Ribbon 来实现的，包括后续我们将要介绍的 Feign，它也是基于 Ribbon 实现的工具。所以，对 Spring Cloud Ribbon 的理解和使用，对于我们使用 Spring Cloud 来构建微服务非常重要。

在这一章中，我们将具体介绍如何使用 Ribbon 来实现客户端的负载均衡，并且通过源码分析来了解 Ribbon 实现客户端负载均衡的基本原理。

客户端负载均衡

负载均衡在系统架构中是一个非常重要，并且是不得不去实施的内容。因为负载均衡是对系统的高可用、网络压力的缓解和处理能力扩容的重要手段之一。我们通常所说的负载均衡都指的是服务端负载均衡，其中分为硬件负载均衡和软件负载均衡。硬件负载均衡主要通过在服务器节点之间安装专门用于负载均衡的设备，比如 F5 等；而软件负载均衡则是通过在服务器上安装一些具有均衡负载功能或模块的软件来完成请求分发工作，比如 Nginx 等。不论采用硬件负载均衡还是软件负载均衡，只要是服务端负载均衡都能以类似下图的架构方式构建起来：



硬件负载均衡的设备或是软件负载均衡的软件模块都会维护一个下挂可用的服务端清单，通过心跳检测来剔除故障的服务端节点以保证清单中都是可以正常访问的服务端节点。当客户端发送请求到负载均衡设备的时候，该设备按某种算法（比如线性轮询、按权重负载、按流量负载等）从维护的可用服务端清单中取出一台服务端的地址，然后进行转发。

而客户端负载均衡和服务端负载均衡最大的不同点在于上面所提到的服务清单所存储的位置。在客户端负载均衡中，所有客户端节点都维护着自己要访问的服务端清单，而这些服务端的清单来自于服务注册中心，比如上一章我们介绍的 Eureka 服务端。同服务端负载均衡的架构类似，在客户端负载均衡中也需要心跳去维护服务端清单的健康性，只是这个步骤需要与服务注册中心配合完成。在 Spring Cloud 实现的服务治理框架中，默认会创建针对各个服务治理框架的 Ribbon 自动化整合配置，比如 Eureka 中的 `org.springframework.cloud.netflix.ribbon.eureka.RibbonEurekaAutoConfiguration`，Consul 中的 `org.springframework.cloud.consul.discovery. RibbonConsulAutoConfiguration`。在实际使用的时候，我们可以通过查看这两个类的实现，以找到它们的配置详情来帮助我们更好地使用它。

通过 Spring Cloud Ribbon 的封装，我们在微服务架构中使用客户端负载均衡调用非常简单，只需要如下两步：

- 服务提供者只需要启动多个服务实例并注册到一个注册中心或是多个相关联的服务注册中心。
- 服务消费者直接通过调用被`@LoadBalanced`注解修饰过的 RestTemplate 来实现面向服务的接口调用。

这样，我们就可以将服务提供者的高可用以及服务消费者的负载均衡调用一起实现了。

RestTemplate 详解

在上一章中，我们已经通过引入 Ribbon 实现了服务消费者的客户端负载均衡功能，读者可以通过查看第 3 章中的“服务发现与消费”一节来获取实验示例。其中，我们使用了一个非常有用的对象 `RestTemplate`。该对象会使用 Ribbon 的自动化配置，同时通过配置`@LoadBalanced` 还能够开启客户端负载均衡。之前我们演示了通过 `RestTemplate` 实现了最简单的服务访问，下面我们将详细介绍 `RestTemplate` 针对几种不同请求类型和参数类型的服务调用实现。

GET 请求

在 `RestTemplate` 中，对 GET 请求可以通过如下两个方法进行调用实现。

第一种：`getForEntity` 函数。该方法返回的是 `ResponseEntity`，该对象是 Spring 对 HTTP 请求响应的封装，其中主要存储了 HTTP 的几个重要元素，比如 HTTP 请求状态码的枚举对象 `HttpStatus`（也就是我们常说的 404、500 这些错误码）、在它的父类 `HttpEntity` 中还存储着 HTTP 请求的头信息对象 `HttpHeaders` 以及泛型类型的请求体对象。比如下面的例子，就是访问 `USER-SERVER` 服务的`/user` 请求，同时最后一个参数 `didi` 会替换 `url` 中的`{1}`占位符，而返回的 `ResponseEntity` 对象中的 body 内容类型会根据第二个参数转换为 String 类型。

```
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<String> responseEntity = restTemplate.getForEntity("http://USER-
SERVICE/user?name={1}", String.class, "didi");
    String body = responseEntity.getBody();
```

若我们希望返回的 body 是一个 User 对象类型，也可以这样实现：

```
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<User> responseEntity = restTemplate.getForEntity("http://USER-
SERVICE/user?name={1}", User.class, "didi");
    User body = responseEntity.getBody();
```

上面的例子是比较常用的方法，`getForEntity` 函数实际上提供了以下三种不同的重载实现。

- getForEntity(String url, Class responseType, Object... urlVariables)：该方法提供了三个参数，其中 url 为请求的地址，responseType 为请求响应体 body 的包装类型，urlVariables 为 url 中的参数绑定。GET 请求的参数绑定通常使用 url 中拼接的方式，比如 http://USER-SERVICE/user?name=didi，我们可以像这样自己将参数拼接到 url 中，但更好的方法是在 url 中使用占位符并配合 urlVariables 参数实现 GET 请求的参数绑定，比如 url 定义为 http://USER-SERVICE/user?name={1}，然后可以这样来调用：getForEntity("http://USER-SERVICE/user?name={1}", String.class, "didi")，其中第三个参数 didi 会替换 url 中的{1}占位符。这里需要注意的是，由于 urlVariables 参数是一个数组，所以它的顺序会对应 url 中占位符定义的数字顺序。
- getForEntity(String url, Class responseType, Map urlVariables)：该方法提供的参数中，只有 urlVariables 的参数类型与上面的方法不同。这里使用了 Map 类型，所以使用该方法进行参数绑定时需要在占位符中指定 Map 中参数的 key 值，比如 url 定义为 http://USER-SERVICE/user?name={name}，在 Map 类型的 urlVariables 中，我们就需要 put 一个 key 为 name 的参数来绑定 url 中{name}占位符的值，比如：

```
RestTemplate restTemplate = new RestTemplate();
Map<String, String> params = new HashMap<>();
params.put("name", "dada");
ResponseEntity<String> responseEntity = restTemplate.getForEntity("http://USER-SERVICE/user?name={name}", String.class, params);
```

- getForEntity(URI url, Class responseType)：该方法使用 URI 对象来替代之前的 url 和 urlVariables 参数来指定访问地址和参数绑定。URI 是 JDK java.net 包下的一个类，它表示一个统一资源标识符（Uniform Resource Identifier）引用。比如下面的例子：

```
RestTemplate restTemplate = new RestTemplate();
UriComponents uriComponents = UriComponentsBuilder.fromUriString(
        "http://USER-SERVICE/user?name={name}")
        .build()
        .expand("dodo")
        .encode();
URI uri = uriComponents.toUri();
ResponseEntity<String> responseEntity = restTemplate.getForEntity(uri, String.class).getBody();
```

更多关于如何定义一个 URI 的方法可以参见 JDK 文档，这里不做详细说明。

第二种：getForObject 函数。该方法可以理解为对 getForEntity 的进一步封装，它通过 HttpMessageConverterExtractor 对 HTTP 的请求响应体 body 内容进行对象转换，实现请求直接返回包装好的对象内容。比如：

```
RestTemplate restTemplate = new RestTemplate();
String result = restTemplate.getForObject(uri, String.class);
```

当 body 是一个 User 对象时，可以直接这样实现：

```
RestTemplate restTemplate = new RestTemplate();
User result = restTemplate.getForObject(uri, User.class);
```

当不需要关注请求响应除 body 外的其他内容时，该函数就非常好用，可以少一个从 Response 中获取 body 的步骤。它与 getForEntity 函数类似，也提供了三种不同的重载实现。

- getForObject(String url, Class responseType, Object... urlVariables)：与 getForEntity 的方法类似，url 参数指定访问的地址，responseType 参数定义该方法的返回类型，urlVariables 参数为 url 中占位符对应的参数。
- getForObject(String url, Class responseType, Map urlVariables)：在该函数中，使用 Map 类型的 urlVariables 替代上面数组形式的 urlVariables，因此使用时在 url 中需要将占位符的名称与 Map 类型中的 key 一一对应设置。
- getForObject(URI url, Class responseType)：该方法使用 URI 对象来替代之前的 url 和 urlVariables 参数使用。

POST 请求

在 RestTemplate 中，对 POST 请求时可以通过如下三个方法进行调用实现。

第一种：postForEntity 函数。该方法同 GET 请求中的 getForEntity 类似，会在调用后返回 ResponseEntity<T>对象，其中 T 为请求响应的 body 类型。比如下面这个例子，使用 postForEntity 提交 POST 请求到 USER-SERVICE 服务的/user 接口，提交的 body 内容为 user 对象，请求响应返回的 body 类型为 String。

```
    RestTemplate restTemplate = new RestTemplate();
    User user = new User("didi", 30);
    ResponseEntity<String> responseEntity =
restTemplate.postForEntity("http://USER-SERVICE/user", user, String.class);
    String body = responseEntity.getBody();
```

postForEntity 函数也实现了三种不同的重载方法。

- postForEntity(String url, Object request, Class responseType, Object... uriVariables)

- postForEntity(String url, Object request, Class responseType, Map uriVariables)
- postForEntity(URI url, Object request, Class responseType)

这些函数中的参数用法大部分与 getForEntity 一致，比如，第一个重载函数和第二个重载函数中的 uriVariables 参数都用来对 url 中的参数进行绑定使用；responseType 参数是对请求响应的 body 内容的类型定义。这里需要注意的是新增加的 request 参数，该参数可以是一个普通对象，也可以是一个 HttpEntity 对象。如果是一个普通对象，而非 HttpEntity 对象的时候，RestTemplate 会将请求对象转换为一个 HttpEntity 对象来处理，其中 Object 就是 request 的类型，request 内容会被视作完整的 body 来处理；而如果 request 是一个 HttpEntity 对象，那么就会被当作一个完成的 HTTP 请求对象来处理，这个 request 中不仅包含了 body 的内容，也包含了 header 的内容。

第二种：postForObject 函数。该方法也跟 getForObject 的类型类似，它的作用是简化 postForEntity 的后续处理。通过直接将请求响应的 body 内容包装成对象来返回使用，比如下面的例子：

```
RestTemplate restTemplate = new RestTemplate();
User user = new User("didi", 20);
String postResult = restTemplate.postForObject("http://USER-SERVICE/user", user, String.class);
```

postForObject 函数也实现了三种不同的重载方法：

- postForObject(String url, Object request, Class responseType, Object... uriVariables)
- postForObject(String url, Object request, Class responseType, Map uriVariables)
- postForObject(URI url, Object request, Class responseType)

这三个函数除了返回的对象类型不同，函数的传入参数均与 postForEntity 一致，因此可参考之前 postForEntity 的说明。

第三种：postForLocation 函数。该方法实现了以 POST 请求提交资源，并返回新资源的 URI，比如下面的例子：

```
User user = new User("didi", 40);
URI responseURI = restTemplate.postForLocation("http://USER-SERVICE/user", user);
```

postForLocation 函数也实现了三种不同的重载方法：

- postForLocation(String url, Object request, Object...urlVariables)

- postForLocation(String url, Object request, Map urlVariables)
- postForLocation(URI url, Object request)

由于 postForLocation 函数会返回新资源的 URI，该 URI 就相当于指定了返回类型，所以此方法实现的 POST 请求不需要像 postForEntity 和 postForObject 那样指定 responseType。其他的参数用法相同。

PUT 请求

在 RestTemplate 中，对 PUT 请求可以通过 put 方法进行调用实现，比如：

```
RestTemplate restTemplate = new RestTemplate();
Long id = 10001L;
User user = new User("didi", 40);
restTemplate.put("http://USER-SERVICE/user/{1}", user, id);
```

put 函数也实现了三种不同的重载方法：

- put(String url, Object request, Object... urlVariables)
- put(String url, Object request, Map urlVariables)
- put(URI url, Object request)

put 函数为 void 类型，所以没有返回内容，也就没有其他函数定义的 responseType 参数，除此之外的其他传入参数定义与用法与 postForObject 基本一致。

DELETE 请求

在 RestTemplate 中，对 DELETE 请求可以通过 delete 方法进行调用实现，比如：

```
RestTemplate restTemplate = new RestTemplate();
Long id = 10001L;
restTemplate.delete("http://USER-SERVICE/user/{1}", id);
```

delete 函数也实现了三种不同的重载方法：

- delete(String url, Object... urlVariables)
- delete(String url, Map urlVariables)
- delete(URI url)

由于我们在进行 REST 请求时，通常都将 DELETE 请求的唯一标识拼接在 url 中，所以 DELETE 请求也不需要 request 的 body 信息，就如上面的三个函数实现一样，非常简单。url 指定 DELETE 请求的位置，urlVariables 绑定 url 中的参数即可。

源码分析

相信很多熟悉 Spring 的读者看到这里一定会产生这样的疑问：RestTemplate 不是 Spring 自己就提供的吗？跟 Ribbon 的客户端负载均衡又有什么关系呢？在本节中，我们将透过现象看本质，探索一下 Ribbon 是如何通过 RestTemplate 实现客户端负载均衡的。

首先，回顾一下之前的消费者示例：我们是如何实现客户端负载均衡的？仔细观察一下之前的实现代码，可以发现在消费者的例子中，可能就@LoadBalanced 这个注解是之前没有接触过的，并且从命名上来看也与负载均衡相关。我们不妨以此为线索来看看 Spring Cloud Ribbon 的源码实现。

从@LoadBalanced 注解源码的注释中可以知道，该注解用来给 RestTemplate 做标记，以使用负载均衡的客户端（LoadBalancerClient）来配置它。

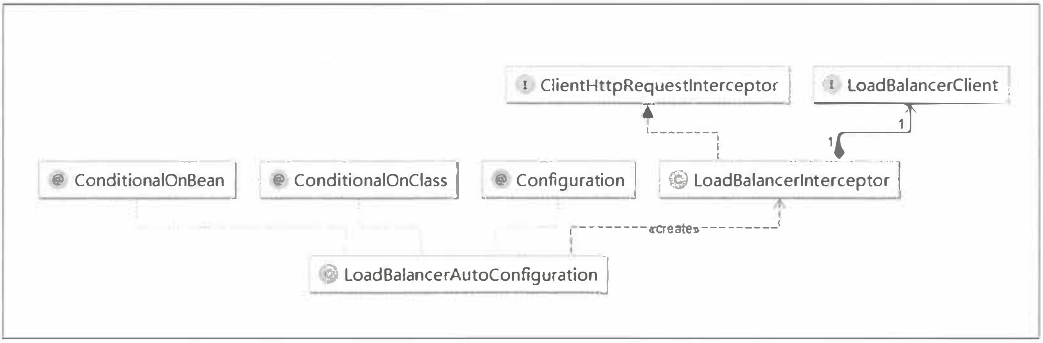
通过搜索 LoadBalancerClient 可以发现，这是 Spring Cloud 中定义的一个接口：

```
public interface LoadBalancerClient {

    ServiceInstance choose(String serviceId);

    <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws
IOException;

    URI reconstructURI(ServiceInstance instance, URI original);

}
```

从该接口中，我们可以通过定义的抽象方法来了解客户端负载均衡器中应具备的几种能力。

- ServiceInstance choose(String serviceId)：根据传入的服务名 serviceId，从负载均衡器中挑选一个对应服务的实例。
- T execute(String serviceId, LoadBalancerRequest request) throws IOException：使用从负载均衡器中挑选出的服务实例来执行请求内容。
- URI reconstructURI(ServiceInstance instance, URI original)：为系统构建一个合适的 host:port 形式的 URI。在分布式系统中，我们使用逻辑上的服务名称作为 host 来构建 URI（替代服务实例的 host:port 形式）进行请求，比如 http://myservice/path/to/service。在该操作的定义中，前者 ServiceInstance 对象是带有 host 和 port 的具体服务实例，而后者 URI 对象则是使用逻辑服务名定义为 host 的 URI，而返回的 URI 内容则是通过 ServiceInstance 的服务实例详情拼接出的具体 host:post 形式的请求地址。

顺着 LoadBalancerClient 接口的所属包 org.springframework.cloud.client.loadbalancer，我们对其内容进行整理，可以得出如下图所示的关系。



从类的命名上可初步判断 LoadBalancerAutoConfiguration 为实现客户端负载均衡器的自动化配置类。通过查看源码，我们可以验证这一点假设：

```
    @Configuration
    @ConditionalOnClass(RestTemplate.class)
    @ConditionalOnBean(LoadBalancerClient.class)
    public class LoadBalancerAutoConfiguration {

        @LoadBalanced
        @Autowired(required = false)
        private List<RestTemplate> restTemplates = Collections.emptyList();

        @Bean
        public SmartInitializingSingleton loadBalancedRestTemplateInitializer(
                final List<RestTemplateCustomizer> customizers) {
            return new SmartInitializingSingleton() {
                @Override
                public void afterSingletonsInstantiated() {
                    for (RestTemplate restTemplate : LoadBalancerAutoConfiguration.
this.restTemplates) {
                        for (RestTemplateCustomizer customizer : customizers) {
                            customizer.customize(restTemplate);
                        }
                    }
                }
            };
        }

        @Bean
        @ConditionalOnMissingBean
```

```
    public RestTemplateCustomizer restTemplateCustomizer(
            final LoadBalancerInterceptor loadBalancerInterceptor) {
        return new RestTemplateCustomizer() {
            @Override
            public void customize(RestTemplate restTemplate) {
                List<ClientHttpRequestInterceptor> list = new ArrayList<>(
                        restTemplate.getInterceptors());
                list.add(loadBalancerInterceptor);
                restTemplate.setInterceptors(list);
            }
        };
    }

    @Bean
    public LoadBalancerInterceptor ribbonInterceptor(
            LoadBalancerClient loadBalancerClient) {
        return new LoadBalancerInterceptor(loadBalancerClient);
    }

}
```

从 LoadBalancerAutoConfiguration 类头上的注解可以知道，Ribbon 实现的负载均衡自动化配置需要满足下面两个条件。

- @ConditionalOnClass(RestTemplate.class)：RestTemplate 类必须存在于当前工程的环境中。
- @ConditionalOnBean(LoadBalancerClient.class)：在 Spring 的 Bean 工程中必须有 LoadBalancerClient 的实现 Bean。

在该自动化配置类中，主要做了下面三件事：

- 创建了一个 LoadBalancerInterceptor 的 Bean，用于实现对客户端发起请求时进行拦截，以实现客户端负载均衡。
- 创建了一个 RestTemplateCustomizer 的 Bean，用于给 RestTemplate 增加 LoadBalancerInterceptor 拦截器。
- 维护了一个被@LoadBalanced 注解修饰的 RestTemplate 对象列表，并在这里进行初始化，通过调用 RestTemplateCustomizer 的实例来给需要客户端负载均衡的 RestTemplate 增加 LoadBalancerInterceptor 拦截器。

接下来，我们看看 LoadBalancerInterceptor 拦截器是如何将一个普通的 RestTemplate 变成客户端负载均衡的：

```
public class LoadBalancerInterceptor implements ClientHttpRequestInterceptor {

    private LoadBalancerClient loadBalancer;

    public LoadBalancerInterceptor(LoadBalancerClient loadBalancer) {
        this.loadBalancer = loadBalancer;
    }

    @Override
    public ClientHttpResponse intercept(final HttpRequest request, final byte[] body,
            final ClientHttpRequestExecution execution) throws IOException {
        final URI originalUri = request.getURI();
        String serviceName = originalUri.getHost();
        return this.loadBalancer.execute(serviceName,
                new LoadBalancerRequest<ClientHttpResponse>() {
                    @Override
                    public ClientHttpResponse apply(final ServiceInstance instance)
                            throws Exception {
                        HttpRequest serviceRequest = new ServiceRequestWrapper(request,
                                instance);
                        return execution.execute(serviceRequest, body);
                    }
                });
    }

    private class ServiceRequestWrapper extends HttpRequestWrapper {

        private final ServiceInstance instance;

        public ServiceRequestWrapper(HttpRequest request, ServiceInstance instance) {
            super(request);
            this.instance = instance;
        }

        @Override
        public URI getURI() {
            URI uri = LoadBalancerInterceptor.this.loadBalancer.reconstructURI(
                    this.instance, getRequest().getURI());
            return uri;
        }
    }
}
```

通过源码以及之前的自动化配置类，我们可以看到在拦截器中注入了 LoadBalancerClient 的实现。当一个被@LoadBalanced 注解修饰的 RestTemplate 对象向外发起 HTTP 请

求时，会被 LoadBalancerInterceptor 类的 intercept 函数所拦截。由于我们在使用 RestTemplate 时采用了服务名作为 host，所以直接从 HttpRequest 的 URI 对象中通过 getHost() 就可以拿到服务名，然后调用 execute 函数去根据服务名来选择实例并发起实际的请求。

分析到这里，LoadBalancerClient 还只是一个抽象的负载均衡器接口，所以我们还需要找到它的具体实现类来进一步进行分析。通过查看 Ribbon 的源码，可以很容易地在 org.springframework.cloud.netflix.ribbon 包下找到对应的实现类 Ribbon-LoadBalancerClient。

```
    public <T> T execute(String serviceId, LoadBalancerRequest<T> request) throws
IOException {
        ILoadBalancer loadBalancer = getLoadBalancer(serviceId);
        Server server = getServer(loadBalancer);
        if (server == null) {
            throw new IllegalStateException("No instances available for " + serviceId);
        }
        RibbonServer ribbonServer = new RibbonServer(serviceId, server,
isSecure(server,
                serviceId), serverIntrospector(serviceId).getMetadata(server));

        RibbonLoadBalancerContext context = this.clientFactory
                .getLoadBalancerContext(serviceId);
        RibbonStatsRecorder statsRecorder = new RibbonStatsRecorder(context, server);

        try {
            T returnVal = request.apply(ribbonServer);
            statsRecorder.recordStats(returnVal);
            return returnVal;
        }
        catch (IOException ex) {
            statsRecorder.recordStats(ex);
            throw ex;
        }
        catch (Exception ex) {
            statsRecorder.recordStats(ex);
            ReflectionUtils.rethrowRuntimeException(ex);
        }
        return null;
    }
```

可以看到，在 execute 函数的实现中，第一步做的就是通过 getServer 根据传入的服务名 serviceId 去获得具体的服务实例：

```
protected Server getServer(ILoadBalancer loadBalancer) {
    if (loadBalancer == null) {
        return null;
    }
    return loadBalancer.chooseServer("default");
}
```

通过 getServer 函数的实现源码，我们可以看到这里获取具体服务实例的时候并没有使用 LoadBalancerClient 接口中的 choose 函数，而是使用了 Netflix Ribbon 自身的 ILoadBalancer 接口中定义的 chooseServer 函数。

我们先来认识一下这个 ILoadBalancer 接口：

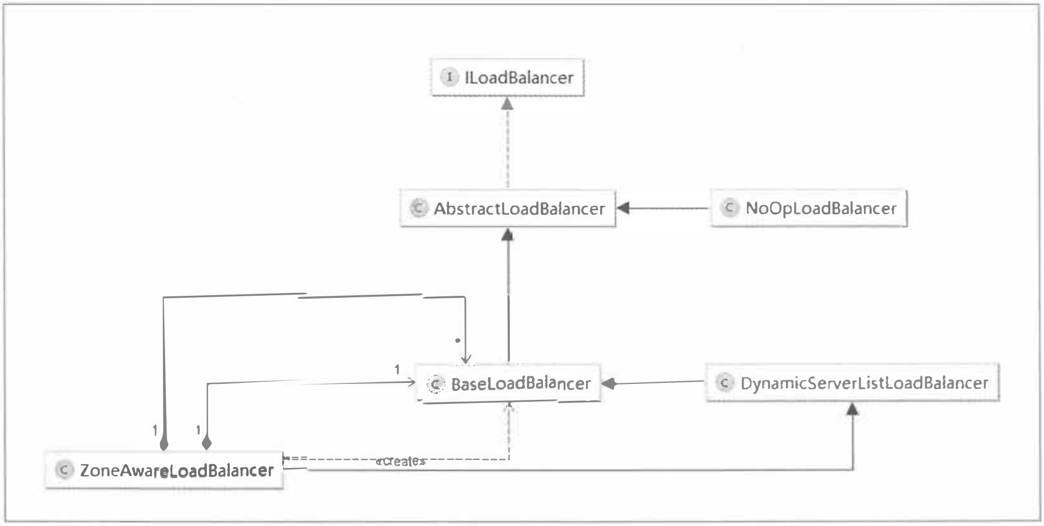
```
public interface ILoadBalancer {

    public void addServers(List<Server> newServers);

    public Server chooseServer(Object key);

    public void markServerDown(Server server);

    public List<Server> getReachableServers();

    public List<Server> getAllServers();
}
```

可以看到，在该接口中定义了一个客户端负载均衡器需要的一系列抽象操作（未列举过期函数）。

- addServers：向负载均衡器中维护的实例列表增加服务实例。
- chooseServer：通过某种策略，从负载均衡器中挑选出一个具体的服务实例。
- markServerDown：用来通知和标识负载均衡器中某个具体实例已经停止服务，不然负载均衡器在下一次获取服务实例清单前都会认为服务实例均是正常服务的。
- getReachableServers：获取当前正常服务的实例列表。
- getAllServers：获取所有已知的服务实例列表，包括正常服务和停止服务的实例。

在该接口定义中涉及的 Server 对象定义是一个传统的服务端节点，在该类中存储了服务端节点的一些元数据信息，包括 host、port 以及一些部署信息等。

而对于该接口的实现，我们整理出如下图所示的结构。可以看到，BaseLoadBalancer 类实现了基础的负载均衡，而 DynamicServerListLoadBalancer 和 ZoneAwareLoadBalancer 在负载均衡的策略上做了一些功能的扩展。



那么在整合 Ribbon 的时候 Spring Cloud 默认采用了哪个具体实现呢？我们通过 `RibbonClientConfiguration` 配置类，可以知道在整合时默认采用了 `ZoneAwareLoadBalancer` 来实现负载均衡器。

```
@Bean
@ConditionalOnMissingBean
public ILoadBalancer ribbonLoadBalancer(IClientConfig config,
        ServerList<Server> serverList, ServerListFilter<Server> serverListFilter,
        IRule rule, IPing ping) {
    ZoneAwareLoadBalancer<Server> balancer = LoadBalancerBuilder.newBuilder()
            .withClientConfig(config).withRule(rule).withPing(ping)
            .withServerListFilter(serverListFilter).withDynamicServerList(serverList)
            .buildDynamicServerListLoadBalancer();
    return balancer;
}
```

下面，我们再回到 `RibbonLoadBalancerClient` 的 `execute` 函数逻辑，在通过 `ZoneAwareLoadBalancer` 的 `chooseServer` 函数获取了负载均衡策略分配到的服务实例对象 `Server` 之后，将其内容包装成 `RibbonServer` 对象（该对象除了存储了服务实例的信息之外，还增加了服务名 `serviceId`、是否需要使用 HTTPS 等其他信息），然后使用该对象再回调 `LoadBalancerInterceptor` 请求拦截器中 `LoadBalancerRequest` 的 `apply(final ServiceInstance instance)` 函数，向一个实际的具体服务实例发起请求，从而实现一开始以服务名为 host 的 URI 请求到 `host:post` 形式的实际访问地址的转换。

在 `apply(final ServiceInstance instance)` 函数中传入的 `ServiceInstance`

接口对象是对服务实例的抽象定义。在该接口中暴露了服务治理系统中每个服务实例需要提供的一些基本信息，比如 serviceId、host、port 等，具体定义如下：

```
public interface ServiceInstance {

    String getServiceId();

    String getHost();

    int getPort();

    boolean isSecure();

    URI getUri();

    Map<String, String> getMetadata();
}
```

而上面提到的具体包装 Server 服务实例的 RibbonServer 对象就是 ServiceInstance 接口的实现，可以看到它除了包含 Server 对象之外，还存储了服务名、是否使用 HTTPS 标识以及一个 Map 类型的元数据集合。

```
protected static class RibbonServer implements ServiceInstance {

    private final String serviceId;
    private final Server server;
    private final boolean secure;
    private Map<String, String> metadata;

    protected RibbonServer(String serviceId, Server server) {
        this(serviceId, server, false, Collections.<String, String> emptyMap());
    }

    protected RibbonServer(String serviceId, Server server, boolean secure,
            Map<String, String> metadata) {
        this.serviceId = serviceId;
        this.server = server;
        this.secure = secure;
        this.metadata = metadata;
    }

    // 省略实现 ServiceInstance 的一些获取 Server 信息的 get 函数
    ...
}
```

那么 apply(final ServiceInstance instance) 函数在接收到了具体

ServiceInstance 实例后，是如何通过 LoadBalancerClient 接口中的 reconstructURI 操作来组织具体请求地址的呢？

```
@Override
public ClientHttpResponse apply(final ServiceInstance instance)
        throws Exception {
    HttpRequest serviceRequest = new ServiceRequestWrapper(request, instance);
    return execution.execute(serviceRequest, body);
}
```

从 apply 的实现中，可以看到它具体执行的时候，还传入了 ServiceRequest-Wrapper 对象，该对象继承了 HttpRequestWrapper 并重写了 getURI 函数，重写后的 getURI 通过调用 LoadBalancerClient 接口的 reconstructURI 函数来重新构建一个 URI 来进行访问。

```
private class ServiceRequestWrapper extends HttpRequestWrapper {

    private final ServiceInstance instance;

    ...

    @Override
    public URI getURI() {
        URI uri = LoadBalancerInterceptor.this.loadBalancer.reconstructURI(
                this.instance, getRequest().getURI());
        return uri;
    }
}
```

在 LoadBalancerInterceptor 拦截器中，ClientHttpRequestExecution 的实例具体执行 execution.execute(serviceRequest, body)时，会调用 Intercepting-ClientHttpRequest 下 InterceptingRequestExecution 类的 execute 函数，具体实现如下：

```
    public ClientHttpResponse execute(HttpRequest request, byte[] body) throws
IOException {
        if (this.iterator.hasNext()) {
            ClientHttpRequestInterceptor nextInterceptor = this.iterator.next();
            return nextInterceptor.intercept(request, body, this);
        }
        else {
            ClientHttpRequest delegate = requestFactory.createRequest(request.getURI(),
request.getMethod());
            delegate.getHeaders().putAll(request.getHeaders());
            if (body.length > 0) {
                StreamUtils.copy(body, delegate.getBody());
```

```
        }
        return delegate.execute();
    }
}
```

可以看到，在创建请求的时候 requestFactory.createRequest(request.getURI(), request.getMethod());，这里的 request.getURI() 会调用之前介绍的 ServiceRequestWrapper 对象中重写的 getURI 函数。此时，它就会使用 RibbonLoadBalancerClient 中实现的 reconstructURI 来组织具体请求的服务实例地址。

```
public URI reconstructURI(ServiceInstance instance, URI original) {
    Assert.notNull(instance, "instance can not be null");
    String serviceId = instance.getServiceId();
    RibbonLoadBalancerContext context = this.clientFactory
            .getLoadBalancerContext(serviceId);
    Server server = new Server(instance.getHost(), instance.getPort());
    boolean secure = isSecure(server, serviceId);
    URI uri = original;
    if (secure) {
        uri = UriComponentsBuilder.fromUri(uri).scheme("https").build().toUri();
    }
    return context.reconstructURIWithServer(server, uri);
}
```

从 reconstructURI 函数中我们可以看到，它通过 ServiceInstance 实例对象的 serviceId，从 SpringClientFactory 类的 clientFactory 对象中获取对应 serviceId 的负载均衡器的上下文 RibbonLoadBalancerContext 对象。然后根据 ServiceInstance 中的信息来构建具体服务实例信息的 Server 对象，并使用 RibbonLoadBalancerContext 对象的 reconstructURIWithServer 函数来构建服务实例的 URI。

为了帮助理解，简单介绍一下上面提到的 SpringClientFactory 和 RibbonLoadBalancerContext：

- SpringClientFactory 类是一个用来创建客户端负载均衡器的工厂类，该工厂类会为每一个不同名的 Ribbon 客户端生成不同的 Spring 上下文。
- RibbonLoadBalancerContext 类是 LoadBalancerContext 的子类，该类用于存储一些被负载均衡器使用的上下文内容和 API 操作（reconstructURIWithServer 就是其中之一）。

从 reconstructURIWithServer 的实现中我们可以看到，它同 reconstructURI 的定义类似。只是 reconstructURI 的第一个保存具体服务实例的参数使用了 Spring Cloud 定义的 ServiceInstance，而 reconstructURIWithServer 中使用了 Netflix

中定义的 Server，所以在 RibbonLoadBalancerClient 实现 reconstructURI 的时候，做了一次转换，使用 ServiceInstance 的 host 和 port 信息构建了一个 Server 对象来给 reconstructURIWithServer 使用。从 reconstructURIWithServer 的实现逻辑中，我们可以看到，它从 Server 对象中获取 host 和 port 信息，然后根据以服务名为 host 的 URI 对象 original 中获取其他请求信息，将两者内容进行拼接整合，形成最终要访问的服务实例的具体地址。

```
public class LoadBalancerContext implements IClientConfigAware {

    ...

    public URI reconstructURIWithServer(Server server, URI original) {
        String host = server.getHost();
        int port = server .getPort();
        if (host.equals(original.getHost())
                && port == original.getPort()) {
            return original;
        }
        String scheme = original.getScheme();
        if (scheme == null) {
            scheme = deriveSchemeAndPortFromPartialUri(original).first();
        }

        try {
            StringBuilder sb = new StringBuilder();
            sb.append(scheme).append("://");
            if (!Strings.isNullOrEmpty(original.getRawUserInfo())) {
                sb.append(original.getRawUserInfo()).append("@");
            }
            sb.append(host);
            if (port >= 0) {
                sb.append(":").append(port);
            }
            sb.append(original.getRawPath());
            if (!Strings.isNullOrEmpty(original.getRawQuery())) {
                sb.append("?").append(original.getRawQuery());
            }
            if (!Strings.isNullOrEmpty(original.getRawFragment())) {
                sb.append("#").append(original.getRawFragment());
            }
            URI newURI = new URI(sb.toString());
            return newURI;
        } catch (URISyntaxException e) {
```

```
                throw new RuntimeException(e);
            }
        }

        ...
    }
```

另外，从 RibbonLoadBalancerClient 的 execute 函数逻辑中，我们还能看到在回调拦截器中，执行具体的请求之后，Ribbon 还通过 RibbonStatsRecorder 对象对服务的请求进行了跟踪记录，这里不再展开说明，有兴趣的读者可以继续研究。

分析到这里，我们已经可以大致理清 Spring Cloud Ribbon 中实现客户端负载均衡的基本脉络，了解了它是如何通过 LoadBalancerInterceptor 拦截器对 RestTemplate 的请求进行拦截，并利用 Spring Cloud 的负载均衡器 LoadBalancerClient 将以逻辑服务名为 host 的 URI 转换成具体的服务实例地址的过程。同时通过分析 LoadBalancerClient 的 Ribbon 实现 RibbonLoadBalancerClient，可以知道在使用 Ribbon 实现负载均衡器的时候，实际使用的还是 Ribbon 中定义的 ILoadBalancer 接口的实现，自动化配置会采用 ZoneAwareLoadBalancer 的实例来实现客户端负载均衡。

负载均衡器

通过之前的分析，我们已经对 Spring Cloud 如何使用 Ribbon 有了基本的了解。虽然 Spring Cloud 中定义了 LoadBalancerClient 作为负载均衡器的通用接口，并且针对 Ribbon 实现了 RibbonLoadBalancerClient，但是它在具体实现客户端负载均衡时，是通过 Ribbon 的 ILoadBalancer 接口实现的。在上一节进行分析时候，我们对该接口的实现结构已经做了一些简单的介绍，下面我们根据 ILoadBalancer 接口的实现类逐个看看它是如何实现客户端负载均衡的。

AbstractLoadBalancer

AbstractLoadBalancer 是 ILoadBalancer 接口的抽象实现。在该抽象类中定义了一个关于服务实例的分组枚举类 ServerGroup，它包含以下三种不同类型。

- ALL：所有服务实例。
- STATUS_UP：正常服务的实例。
- STATUS_NOT_UP：停止服务的实例。

另外，还实现了一个 chooseServer() 函数，该函数通过调用接口中的 chooseServer(Object key) 实现，其中参数 key 为 null，表示在选择具体服务实例时忽略 key 的条件判断。

最后，还定义了两个抽象函数。

- getServerList(ServerGroup serverGroup)：定义了根据分组类型来获取不同的服务实例的列表。
- getLoadBalancerStats()：定义了获取 LoadBalancerStats 对象的方法，LoadBalancerStats 对象被用来存储负载均衡器中各个服务实例当前的属性和统计信息。这些信息非常有用，我们可以利用这些信息来观察负载均衡器的运行情况，同时这些信息也是用来制定负载均衡策略的重要依据。

```
public abstract class AbstractLoadBalancer implements ILoadBalancer {

    public enum ServerGroup{
        ALL,
        STATUS_UP,
        STATUS_NOT_UP
    }

    public Server chooseServer() {
        return chooseServer(null);
    }

    public abstract List<Server> getServerList(ServerGroup serverGroup);

    public abstract LoadBalancerStats getLoadBalancerStats();
}
```

BaseLoadBalancer

BaseLoadBalancer 类是 Ribbon 负载均衡器的基础实现类，在该类中定义了很多关于负载均衡器相关的基础内容。

- 定义并维护了两个存储服务实例 Server 对象的列表。一个用于存储所有服务实例的清单，一个用于存储正常服务的实例清单。

```
@Monitor(name = PREFIX + "AllServerList", type = DataSourceType.INFORMATIONAL)
protected volatile List<Server> allServerList = Collections
    .synchronizedList(new ArrayList<Server>());
@Monitor(name = PREFIX + "UpServerList", type = DataSourceType.INFORMATIONAL)
protected volatile List<Server> upServerList = Collections
    .synchronizedList(new ArrayList<Server>());
```

- 定义了之前我们提到的用来存储负载均衡器各服务实例属性和统计信息的 LoadBalancerStats 对象。

- 定义了检查服务实例是否正常服务的 IPing 对象，在 BaseLoadBalancer 中默认为 null，需要在构造时注入它的具体实现。
- 定义了检查服务实例操作的执行策略对象 IPingStrategy，在 BaseLoadBalancer 中默认使用了该类中定义的静态内部类 SerialPingStrategy 实现。根据源码，我们可以看到该策略采用线性遍历 ping 服务实例的方式实现检查。该策略在当 IPing 的实现速度不理想，或是 Server 列表过大时，可能会影响系统性能，这时候需要通过实现 IPingStrategy 接口并重写 pingServers(IPing ping, Server[] servers) 函数去扩展 ping 的执行策略。

```
private static class SerialPingStrategy implements IPingStrategy {
    @Override
    public boolean[] pingServers(IPing ping, Server[] servers) {
        int numCandidates = servers.length;
        boolean[] results = new boolean[numCandidates];

        if (logger.isDebugEnabled()) {
            logger.debug("LoadBalancer:  PingTask executing ["
                        + numCandidates + "] servers configured");
        }

        for (int i = 0; i < numCandidates; i++) {
            results[i] = false;
            try {
                if (ping != null) {
                    results[i] = ping.isAlive(servers[i]);
                }
            } catch (Throwable t) {
                logger.error("Exception while pinging Server:"
                        + servers[i], t);
            }
        }
        return results;
    }
}
```

- 定义了负载均衡的处理规则 IRule 对象，从 BaseLoadBalancer 中 chooseServer(Object key) 的实现源码，我们可以知道，负载均衡器实际将服务实例选择任务委托给了 IRule 实例中的 choose 函数来实现。而在这里，默认初始化了 RoundRobinRule 为 IRule 的实现对象。RoundRobinRule 实现了最基本且常用的线性负载均衡规则。

```
public Server chooseServer(Object key) {
    if (counter == null) {
```

```
            counter = createCounter();
        }
        counter.increment();
        if (rule == null) {
            return null;
        } else {
            try {
                return rule.choose(key);
            } catch (Throwable t) {
                return null;
            }
        }
}
```

- 启动 ping 任务：在 BaseLoadBalancer 的默认构造函数中，会直接启动一个用于定时检查 Server 是否健康的任务。该任务默认的执行间隔为 10 秒。
- 实现了 ILoadBalancer 接口定义的负载均衡器应具备以下一系列基本操作。
    - addServers(List newServers)：向负载均衡器中增加新的服务实例列表，该实现将原本已经维护着的所有服务实例清单 allServerList 和新传入的服务实例清单 newServers 都加入到 newList 中，然后通过调用 setServersList 函数对 newList 进行处理，在 BaseLoadBalancer 中实现的时候会使用新的列表覆盖旧的列表。而之后介绍的几个扩展实现类对于服务实例清单更新的优化都是通过对 setServersList 函数的重写来实现的。

```
public void addServers(List<Server> newServers) {
if (newServers != null && newServers.size() > 0) {
    try {
        ArrayList<Server> newList = new ArrayList<Server>();
        newList.addAll(allServerList);
        newList.addAll(newServers);
        setServersList(newList);
    } catch (Exception e) {
        logger.error("Exception while adding Servers", e);
    }
}
}
```

- chooseServer(Object key)：挑选一个具体的服务实例，在上面介绍 IRule 的时候，已经做了说明，这里不再赘述。
- markServerDown(Server server)：标记某个服务实例暂停服务。

```
public void markServerDown(Server server) {
    if (server == null) {
```

```
        return;
    }
    if (!server.isAlive()) {
        return;
    }
    logger.error("LoadBalancer:  markServerDown called on ["
            + server.getId() + "]");
    server.setAlive(false);
    notifyServerStatusChangeListener(singleton(server));
}
```

- getReachableServers()：获取可用的服务实例列表。由于 BaseLoadBalancer 中单独维护了一个正常服务的实例清单，所以直接返回即可。

```
public List<Server> getReachableServers() {
    return Collections.unmodifiableList(upServerList);
}
```

- getAllServers()：获取所有的服务实例列表。由于 BaseLoadBalancer 中单独维护了一个所有服务的实例清单，所以也直接返回它即可。

```
public List<Server> getAllServers() {
    return Collections.unmodifiableList(allServerList);
}
```

DynamicServerListLoadBalancer

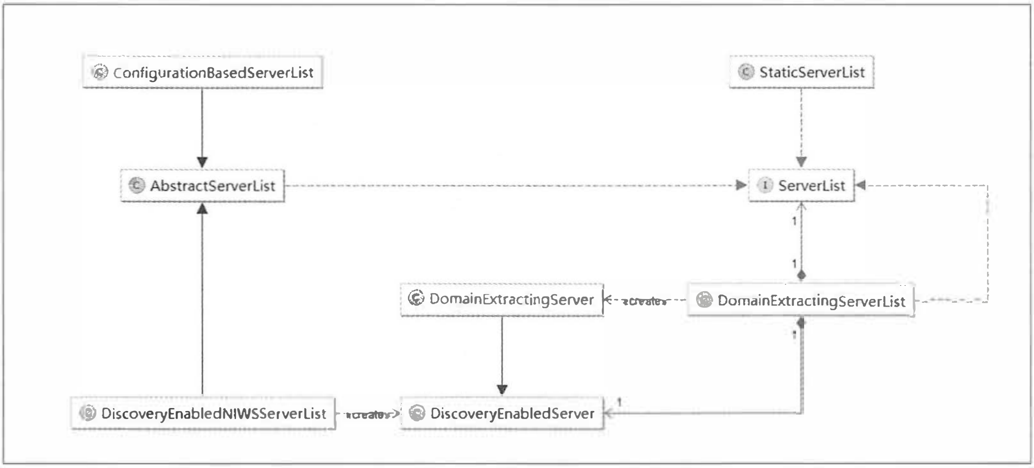
DynamicServerListLoadBalancer 类继承于 BaseLoadBalancer 类，它是对基础负载均衡器的扩展。在该负载均衡器中，实现了服务实例清单在运行期的动态更新能力；同时，它还具备了对服务实例清单的过滤功能，也就是说，我们可以通过过滤器来选择性地获取一批服务实例清单。下面我们具体来看看在该类中增加了一些什么内容。

ServerList

从 DynamicServerListLoadBalancer 的成员定义中，我们马上可以发现新增了一个关于服务列表的操作对象 ServerList<T> serverListImpl。其中泛型 T 从类名中对于 T 的限定 DynamicServerListLoadBalancer<T extends Server>可以获知它是一个 Server 的子类，即代表了一个具体的服务实例的扩展类。而 ServerList 接口定义如下所示：

```
public interface ServerList<T extends Server> {

    public List<T> getInitialListOfServers();

    public List<T> getUpdatedListOfServers();
}
```

它定义了两个抽象方法：getInitialListOfServers 用于获取初始化的服务实例清单，而 getUpdatedListOfServers 用于获取更新的服务实例清单。那么该接口的实现有哪些呢？通过搜索源码，我们可以整理出如下图所示的结构。



从上图中我们可以看到有多个 ServerList 的实现类，那么在 DynamicServerListLoadBalancer 中的 ServerList 默认配置到底使用了哪个具体实现呢？既然在该负载均衡器中需要实现服务实例的动态更新，那么势必需要 Ribbon 具备访问 Eureka 来获取服务实例的能力，所以我们从 Spring Cloud 整合 Ribbon 与 Eureka 的包 org.springframework.cloud.netflix.ribbon.eureka 下进行探索，可以找到配置类 EurekaRibbonClientConfiguration，在该类中可以找到如下创建 ServerList 实例的内容：

```
    @Bean
    @ConditionalOnMissingBean
    public ServerList<?> ribbonServerList(IClientConfig config) {
        DiscoveryEnabledNIWSServerList discoveryServerList = new
DiscoveryEnabledNIWSServerList(
                config);
        DomainExtractingServerList serverList = new DomainExtractingServerList(
                discoveryServerList, config, this.approximateZoneFromHostname);
        return serverList;
    }
```

可以看到，这里创建的是一个 DomainExtractingServerList 实例，从下面它的源码中我们可以看到，在它内部还定义了一个 ServerList list。同时，DomainExtractingServerList 类中对 getInitialListOfServers 和 getUpdatedListOfServers 的具体实现，其实委托给了内部定义的 ServerList list 对象，而该

对象是通过创建 DomainExtractingServerList 时，由构造函数传入的 DiscoveryEnabledNIWSServerList 实现的。

```
public class DomainExtractingServerList implements ServerList<DiscoveryEnabledServer> {

    private ServerList<DiscoveryEnabledServer> list;
    private IClientConfig clientConfig;
    private boolean approximateZoneFromHostname;

    public DomainExtractingServerList(ServerList<DiscoveryEnabledServer> list,
            IClientConfig clientConfig, boolean approximateZoneFromHostname) {
        this.list = list;
        this.clientConfig = clientConfig;
        this.approximateZoneFromHostname = approximateZoneFromHostname;
    }

    @Override
    public List<DiscoveryEnabledServer> getInitialListOfServers() {
        List<DiscoveryEnabledServer> servers = setZones(this.list
                .getInitialListOfServers());
        return servers;
    }

    @Override
    public List<DiscoveryEnabledServer> getUpdatedListOfServers() {
        List<DiscoveryEnabledServer> servers = setZones(this.list
                .getUpdatedListOfServers());
        return servers;
    }
    ...
}
```

那么 DiscoveryEnabledNIWSServerList 是如何实现这两个服务实例获取的呢？我们从源码中可以看到这两个方法都是通过该类中的一个私有函数 obtainServersViaDiscovery 通过服务发现机制来实现服务实例的获取的。

```
@Override
public List<DiscoveryEnabledServer> getInitialListOfServers(){
    return obtainServersViaDiscovery();
}

@Override
public List<DiscoveryEnabledServer> getUpdatedListOfServers(){
    return obtainServersViaDiscovery();
}
```

而 obtainServersViaDiscovery 的实现逻辑如下所示，主要依靠 EurekaClient 从服务注册中心中获取到具体的服务实例 InstanceInfo 列表（EurekaClient 的具体实现，我们在分析 Eureka 的源码时已经做了详细的介绍，这里传入的 vipAddress 可以理解为逻辑上的服务名，比如 USER-SERVICE）。接着，对这些服务实例进行遍历，将状态为 UP（正常服务）的实例转换成 DiscoveryEnabledServer 对象，最后将这些实例组织成列表返回。

```
    private List<DiscoveryEnabledServer> obtainServersViaDiscovery() {
        List<DiscoveryEnabledServer> serverList = new
ArrayList<DiscoveryEnabledServer>();

        if (eurekaClientProvider == null || eurekaClientProvider.get() == null) {
            logger.warn("EurekaClient has not been initialized yet, returning an empty
list");
            return new ArrayList<DiscoveryEnabledServer>();
        }

        EurekaClient eurekaClient = eurekaClientProvider.get();
        if (vipAddresses!=null){
            for (String vipAddress : vipAddresses.split(",")) {
                List<InstanceInfo> listOfInstanceInfo =
eurekaClient.getInstancesByVipAddress(
                        vipAddress, isSecure, targetRegion);
                for (InstanceInfo ii : listOfInstanceInfo) {
                    if (ii.getStatus().equals(InstanceStatus.UP)) {
                        // 省略了一些实例信息的加工逻辑
                        DiscoveryEnabledServer des = new DiscoveryEnabledServer(ii,
isSecure, shouldUseIpAddr);
                        des.setZone(DiscoveryClient.getZone(ii));
                        serverList.add(des);
                    }
                }
                if (serverList.size()>0 && prioritizeVipAddressBasedServers){
                    break;
                }
            }
        }
        return serverList;
    }
```

在 DiscoveryEnabledNIWSServerList 中通过 EurekaClient 从服务注册中心获取到最新的服务实例清单后，返回的 List 到了 DomainExtractingServerList 类中，将继续通过 setZones 函数进行处理。而这里的处理具体内容如下所示，主要完成将 DiscoveryEnabledNIWSServerList 返回的 List 列表中的元素，转换成内部定义的

DiscoveryEnabledServer 的子类对象 DomainExtractingServer，在该对象的构造函数中将为服务实例对象设置一些必要的属性，比如 id、zone、isAliveFlag、readyToServe 等信息。

```
    private List<DiscoveryEnabledServer> setZones(List<DiscoveryEnabledServer>
servers) {
        List<DiscoveryEnabledServer> result = new ArrayList<>();
        boolean isSecure = this.clientConfig.getPropertyAsBoolean(
                CommonClientConfigKey.IsSecure, Boolean.TRUE);
        boolean shouldUseIpAddr = this.clientConfig.getPropertyAsBoolean(
                CommonClientConfigKey.UseIPAddrForServer, Boolean.FALSE);
        for (DiscoveryEnabledServer server : servers) {
            result.add(new DomainExtractingServer(server, isSecure, shouldUseIpAddr,
                    this.approximateZoneFromHostname));
        }
        return result;
    }
```

ServerListUpdater

通过上面的分析我们已经知道了 Ribbon 与 Eureka 整合后，如何实现从 Eureka Server 中获取服务实例清单。那么它又是如何触发向 Eureka Server 去获取服务实例清单以及如何在获取到服务实例清单后更新本地的服务实例清单的呢？继续来看 DynamicServer-ListLoadBalancer 中的实现内容，我们可以很容易地找到下面定义的关于 ServerListUpdater 的内容：

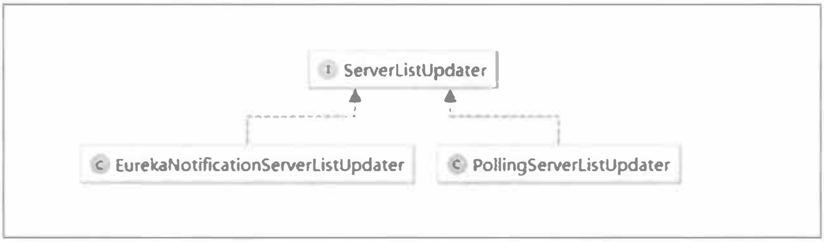
```
    protected final ServerListUpdater.UpdateAction updateAction = new
ServerListUpdater.UpdateAction() {
        @Override
        public void doUpdate() {
            updateListOfServers();
        }
    };

    protected volatile ServerListUpdater serverListUpdater;
```

根据该接口的命名，我们基本就能猜到，这个对象实现的是对 ServerList 的更新，所以可以称它为“服务更新器”。从下面的源码中可以看到，在 ServerListUpdater 内部还定义了一个 UpdateAction 接口，上面定义的 updateAction 对象就是以匿名内部类的方式创建了一个它的具体实现，其中 doUpdate 实现的内容就是对 ServerList 的具体更新操作。除此之外，“服务更新器”中还定义了一系列控制它和获取它的信息的操作。

```
    public interface ServerListUpdater {

        public interface UpdateAction {
```

```
        void doUpdate();
    }

    // 启动服务更新器，传入的 UpdateAction 对象为更新操作的具体实现。
    void start(UpdateAction updateAction);

    // 停止服务更新器
    void stop();

    // 获取最近的更新时间戳
    String getLastUpdate();

    // 获取上一次更新到现在的时间间隔，单位为毫秒
    long getDurationSinceLastUpdateMs();

    // 获取错过的更新周期数
    int getNumberMissedCycles();

    // 获取核心线程数
    int getCoreThreads();
}
```

而 ServerListUpdater 的实现类不多，具体如下图所示。



根据两个类的注释，我们可以很容易地知道它们的作用。

- PollingServerListUpdater：动态服务列表更新的默认策略，也就是说，DynamicServerListLoadBalancer 负载均衡器中的默认实现就是它，它通过定时任务的方式进行服务列表的更新。
- EurekaNotificationServerListUpdater：该更新器也可服务于 Dynamic-ServerListLoadBalancer 负载均衡器，但是它的触发机制与 PollingServer-ListUpdater 不同，它需要利用 Eureka 的事件监听器来驱动服务列表的更新操作。

下面我们来详细看看它默认实现的 PollingServerListUpdater。先从用于启动“服务更新器”的 start 函数源码看起，具体如下。我们可以看到 start 函数的实现内容

验证了之前提到的：以定时任务的方式进行服务列表的更新。它先创建了一个 Runnable 的线程实现，在该实现中调用了上面提到的具体更新服务实例列表的方法 updateAction.doUpdate()，最后再为这个 Runnable 线程实现启动了一个定时任务来执行。

```
@Override
public synchronized void start(final UpdateAction updateAction) {
    if (isActive.compareAndSet(false, true)) {
        final Runnable wrapperRunnable = new Runnable() {
            @Override
            public void run() {
                if (!isActive.get()) {
                    if (scheduledFuture != null) {
                        scheduledFuture.cancel(true);
                    }
                    return;
                }
                try {
                    updateAction.doUpdate();
                    lastUpdated = System.currentTimeMillis();
                } catch (Exception e) {
                    logger.warn("Failed one update cycle", e);
                }
            }
        };

        scheduledFuture = getRefreshExecutor().scheduleWithFixedDelay(
                wrapperRunnable,
                initialDelayMs,
                refreshIntervalMs,
                TimeUnit.MILLISECONDS
        );
    } else {
        logger.info("Already active, no-op");
    }
}
```

继续看 Polling ServerListUpdater 的其他内容，我们可以找到用于启动定时任务的两个重要参数 initialDelayMs 和 refreshIntervalMs 的默认定义分别为 1000 和 30*1000，单位为毫秒。也就是说，更新服务实例在初始化之后延迟 1 秒后开始执行，并以 30 秒为周期重复执行。除了这些内容之外，还能看到它还会记录最后更新时间、是否存活等信息，同时也实现了 ServerListUpdater 中定义的一些其他操作内容，这些操作相对比较简单，这里不再具体说明，有兴趣的读者可以自己查看源码了解其实现原理。

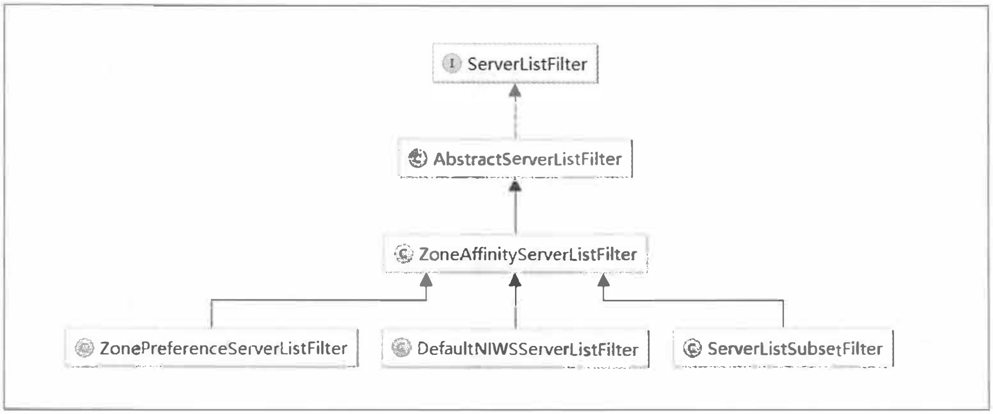
ServerListFilter

在了解了更新服务实例的定时任务是如何启动的之后，我们回到 updateAction.doUpdate()调用的具体实现位置，在 DynamicServerListLoadBalancer 中，它的实际实现委托给了 updateListOfServers 函数，具体实现如下：

```
public void updateListOfServers() {
    List<T> servers = new ArrayList<T>();
    if (serverListImpl != null) {
        servers = serverListImpl.getUpdatedListOfServers();
        LOGGER.debug("List of Servers for {} obtained from Discovery client: {}",
                getIdentifier(), servers);

        if (filter != null) {
            servers = filter.getFilteredListOfServers(servers);
            LOGGER.debug("Filtered List of Servers for {} obtained from Discovery
client: {}",
                    getIdentifier(), servers);
        }
    }
    updateAllServerList(servers);
}
```

可以看到，这里终于用到了之前提到的 ServerList 的 getUpdatedListOfServers()，通过之前的介绍已经知道这一步实现了从 Eureka Server 中获取服务可用实例的列表。在获得了服务实例列表之后，这里又将引入一个新的对象 filter，追溯该对象的定义，我们可以找到它是 ServerListFilter 定义的。

ServerListFilter 接口非常简单，该接口中定义了一个方法 List getFilteredListOfServers(List servers)，主要用于实现对服务实例列表的过滤，通过传入的服务实例清单，根据一些规则返回过滤后的服务实例清单。该接口的实现如下图所示。



其中，除了 ZonePreferenceServerListFilter 的实现是 Spring Cloud Ribbon 中对 Netflix Ribbon 的扩展实现外，其他均是 Netflix Ribbon 中的原生实现类。下面，我们可以分别看看这些过滤器实现都有什么特点。

- AbstractServerListFilter：这是一个抽象过滤器，在这里定义了过滤时需要的一个重要依据对象 LoadBalancerStats，我们在之前介绍过，该对象存储了关于负载均衡器的一些属性和统计信息等。

```
    public abstract class AbstractServerListFilter<T extends Server> implements
ServerListFilter<T> {

        private volatile LoadBalancerStats stats;

        public void setLoadBalancerStats(LoadBalancerStats stats) {
            this.stats = stats;
        }

        public LoadBalancerStats getLoadBalancerStats() {
            return stats;
        }
    }
```

- ZoneAffinityServerListFilter：该过滤器基于“区域感知（Zone Affinity）”的方式实现服务实例的过滤，也就是说，它会根据提供服务的实例所处的区域（Zone）与消费者自身的所处区域（Zone）进行比较，过滤掉那些不是同处一个区域的实例。

```
    public List<T> getFilteredListOfServers(List<T> servers) {
        if (zone != null && (zoneAffinity || zoneExclusive) && servers !=null &&
servers.size() > 0){
            List<T> filteredServers = Lists.newArrayList(Iterables.filter(
                    servers, this.zoneAffinityPredicate.getServerOnlyPredicate()));
            if (shouldEnableZoneAffinity(filteredServers)) {
                return filteredServers;
            } else if (zoneAffinity) {
                overrideCounter.increment();
            }
        }
        return servers;
    }
```

从上面的源码中我们可以看到，对于服务实例列表的过滤是通过 Iterables.filter(servers,this.zoneAffinityPredicate.getServerOnlyPredicate()) 来实现的，其中判断依据由 ZoneAffinityPredicate 实现服务实例与消费者的 Zone

比较。而在过滤之后，这里并不会马上返回过滤的结果，而是通过 shouldEnableZoneAffinity 函数来判断是否要启用“区域感知”的功能。从下面 shouldEnableZoneAffinity 的实现中，我们可以看到，它使用了 LoadBalancerStats 的 getZoneSnapshot 方法来获取这些过滤后的同区域实例的基础指标（包含实例数量、断路器断开数、活动请求数、实例平均负载等），根据一系列的算法求出下面的几个评价值并与设置的阈值进行对比（下面的为默认值），若有一个条件符合，就不启用“区域感知”过滤的服务实例清单。这一算法实现为集群出现区域故障时，依然可以依靠其他区域的实例进行正常服务提供了完善的高可用保障。同时，通过这里的介绍，我们也可以关联着来理解之前介绍 Eureka 的时候提到的对于区域分配设计来保证跨区域故障的高可用问题。

- blackOutServerPercentage：故障实例百分比（断路器断开数/实例数量）>= 0.8。
- activeReqeustsPerServer：实例平均负载 >= 0.6。
- availableServers：可用实例数（实例数量 - 断路器断开数）< 2。

```
    private boolean shouldEnableZoneAffinity(List<T> filtered) {
        if (!zoneAffinity && !zoneExclusive) {
            return false;
        }
        if (zoneExclusive) {
            return true;
        }
        LoadBalancerStats stats = getLoadBalancerStats();
        if (stats == null) {
            return zoneAffinity;
        } else {
            logger.debug("Determining if zone affinity should be enabled with given server
list: {}", filtered);
            ZoneSnapshot snapshot = stats.getZoneSnapshot(filtered);
            double loadPerServer = snapshot.getLoadPerServer();
            int instanceCount = snapshot.getInstanceCount();
            int circuitBreakerTrippedCount = snapshot.getCircuitTrippedCount();
            if (((double) circuitBreakerTrippedCount) / instanceCount >=
blackOutServerPercentageThreshold.get()
                || loadPerServer >= activeReqeustsPerServerThreshold.get()
                || (instanceCount - circuitBreakerTrippedCount) <
availableServersThreshold.get()) {
                logger.debug("zoneAffinity is overriden. blackOutServerPercentage: {},
activeReqeustsPerServer: {}, availableServers: {}",
                new Object[] {(double) circuitBreakerTrippedCount / instanceCount,
loadPerServer, instanceCount - circuitBreakerTrippedCount});
                return false;
```

```
        } else {
            return true;
        }
    }
}
```

- `DefaultNIWSServerListFilter`：该过滤器完全继承自 `ZoneAffinity-ServerListFilter`，是默认的 NIWS（Netflix Internal Web Service）过滤器。
- `ServerListSubsetFilter`：该过滤器也继承自 `ZoneAffinityServer-ListFilter`，它非常适用于拥有大规模服务器集群（上百或更多）的系统。因为它可以产生一个“区域感知”结果的子集列表，同时它还能够通过比较服务实例的通信失败数量和并发连接数来判定该服务是否健康来选择性地从服务实例列表中剔除那些相对不够健康的实例。该过滤器的实现主要分为以下三步：

  1. 获取“区域感知”的过滤结果，作为候选的服务实例清单。
  2. 从当前消费者维护的服务实例子集中剔除那些相对不够健康的实例（同时也将这些实例从候选清单中剔除，防止第三步的时候又被选入），不够健康的标准如下所示。

     a. 服务实例的并发连接数超过客户端配置的值，默认为 `0`，配置参数为 `<clientName>.<nameSpace>.ServerListSubsetFilter.eliminationConnectionThresold`。

     b. 服务实例的失败数超过客户端配置的值，默认为 `0`，配置参数为 `<clientName>.<nameSpace>.ServerListSubsetFilter.eliminationFailureThresold`。

     c. 如果按符合上面任一规则的服务实例剔除后，剔除比例小于客户端默认配置的百分比，默认为 `0.1`（10%），配置参数为`<clientName>.<nameSpace>.ServerListSubsetFilter.forceEliminatePercent`，那么就先对剩下的实例列表进行健康排序，再从最不健康的实例进行剔除，直到达到配置的剔除百分比。

  3. 在完成剔除后，清单已经少了至少 10%（默认值）的服务实例，最后通过随机的方式从候选清单中选出一批实例加入到清单中，以保持服务实例子集与原来的数量一致，而默认的实例子集数量为 `20`，其配置参数为`<clientName>.<nameSpace>.ServerListSubsetFilter.size`。

- `ZonePreferenceServerListFilter`：Spring Cloud 整合时新增的过滤器。若使用 Spring Cloud 整合 Eureka 和 Ribbon 时会默认使用该过滤器。它实现了通过配

置或者 Eureka 实例元数据的所属区域（Zone）来过滤出同区域的服务实例。如下面的源码所示，它的实现非常简单，首先通过父类 ZoneAffinityServerListFilter 的过滤器来获得“区域感知”的服务实例列表，然后遍历这个结果，取出根据消费者配置预设的区域 Zone 来进行过滤，如果过滤的结果是空就直接返回父类获取的结果，如果不为空就返回通过消费者配置的 Zone 过滤后的结果。

```
@Override
public List<Server> getFilteredListOfServers(List<Server> servers) {
    List<Server> output = super.getFilteredListOfServers(servers);
    if (this.zone != null && output.size() == servers.size()) {
        List<Server> local = new ArrayList<Server>();
        for (Server server : output) {
            if (this.zone.equalsIgnoreCase(server.getZone())) {
                local.add(server);
            }
        }
        if (!local.isEmpty()) {
            return local;
        }
    }
    return output;
}
```

ZoneAwareLoadBalancer

ZoneAwareLoadBalancer 负载均衡器是对 DynamicServerListLoadBalancer 的扩展。在 DynamicServerListLoadBalancer 中，我们可以看到它并没有重写选择具体服务实例的 chooseServer 函数，所以它依然会采用在 BaseLoadBalancer 中实现的算法。使用 RoundRobinRule 规则，以线性轮询的方式来选择调用的服务实例，该算法实现简单并没有区域（Zone）的概念，所以它会把所有实例视为一个 Zone 下的节点来看待，这样就会周期性地产生跨区域（Zone）访问的情况，由于跨区域会产生更高的延迟，这些实例主要以防止区域性故障实现高可用为目的而不能作为常规访问的实例，所以在多区域部署的情况下会有一定的性能问题，而该负载均衡器则可以避免这样的问题。那么它是如何实现的呢？

首先，在 ZoneAwareLoadBalancer 中，我们可以发现，它并没有重写 setServersList，说明实现服务实例清单的更新主逻辑没有修改。但是我们可以发现它重写了这个函数 setServerListForZones(Map<String, List<Server>>zoneServersMap)。看到这里可能会有一些陌生，因为它并不是接口中定义的必备函数，所以我们不妨去父类 DynamicServerListLoadBalancer 中寻找一下该函数，我们可以找到下面的定义：

```
public void setServersList(List lsrv) {
    super.setServersList(lsrv);
    List<T> serverList = (List<T>) lsrv;
    Map<String, List<Server>> serversInZones = new HashMap<String, List<Server>>();
    ...
    setServerListForZones(serversInZones);
}

protected void setServerListForZones(Map<String, List<Server>> zoneServersMap) {
    LOGGER.debug("Setting server list for zones: {}", zoneServersMap);
    getLoadBalancerStats().updateZoneServerMapping(zoneServersMap);
}
```

setServerListForZones 函数的调用位于更新服务实例清单函数 setServersList 的最后，同时从其实现的内容来看，它在父类 DynamicServerListLoadBalancer 中的作用是根据按区域 Zone 分组的实例列表，为负载均衡器中的 LoadBalancerStats 对象创建 ZoneStats 并放入 Map zoneStatsMap 集合中，每一个区域 Zone 对应一个 ZoneStats，它用于存储每个 Zone 的一些状态和统计信息。

在 ZoneAwareLoadBalancer 中对 setServerListForZones 的重写如下：

```
protected void setServerListForZones(Map<String, List<Server>> zoneServersMap) {
    super.setServerListForZones(zoneServersMap);
    if (balancers == null) {
        balancers = new ConcurrentHashMap<String, BaseLoadBalancer>();
    }
    for (Map.Entry<String, List<Server>> entry: zoneServersMap.entrySet()) {
        String zone = entry.getKey().toLowerCase();
        getLoadBalancer(zone).setServersList(entry.getValue());
    }
    for (Map.Entry<String, BaseLoadBalancer> existingLBEntry: balancers.entrySet()) {
        if (!zoneServersMap.keySet().contains(existingLBEntry.getKey())) {
            existingLBEntry.getValue().setServersList(Collections.emptyList());
        }
    }
}
```

可以看到，在该实现中创建了一个 ConcurrentHashMap()类型的 balancers 对象，它将用来存储每个 Zone 区域对应的负载均衡器。而具体的负载均衡器的创建则是通过在下面的第一个循环中调用 getLoadBalancer 函数来完成，同时在创建负载均衡器的时候会创建它的规则（如果当前实现中没有 IRule 的实例，就创建一个 AvailabilityFilteringRule 规则；如果已经有具体实例，就克隆一个）。在创建完负载均衡器后又马上调用 setServersList 函数为其设置对应 Zone 区域的实例清单。而第二个循环则是对 Zone 区域中实例清单的检查，看看是否有 Zone 区域下已经没有实例了，是的话就将

balancers 中对应 Zone 区域的实例列表清空，该操作的作用是为了后续选择节点时，防止过时的 Zone 区域统计信息干扰具体实例的选择算法。

在了解了该负载均衡器是如何扩展服务实例清单的实现后，我们来具体看看它是如何挑选服务实例，来实现对区域的识别的：

```
    public Server chooseServer(Object key) {
        if (!ENABLED.get() || getLoadBalancerStats().getAvailableZones().size() <= 1) {
            logger.debug("Zone aware logic disabled or there is only one zone");
            return super.chooseServer(key);
        }
        Server server = null;
        try {
            LoadBalancerStats lbStats = getLoadBalancerStats();
            Map<String, ZoneSnapshot> zoneSnapshot = ZoneAvoidanceRule.createSnapshot
(lbStats);
            logger.debug("Zone snapshots: {}", zoneSnapshot);
            ...
            Set<String> availableZones = ZoneAvoidanceRule.getAvailableZones
(zoneSnapshot, triggeringLoad.get(), triggeringBlackoutPercentage.get());
            logger.debug("Available zones: {}", availableZones);
            if (availableZones != null &&  availableZones.size() <
zoneSnapshot.keySet().size()) {
                String zone = ZoneAvoidanceRule.randomChooseZone(zoneSnapshot,
availableZones);
                logger.debug("Zone chosen: {}", zone);
                if (zone != null) {
                    BaseLoadBalancer zoneLoadBalancer = getLoadBalancer(zone);
                    server = zoneLoadBalancer.chooseServer(key);
                }
            }
        } catch (Throwable e) {
            logger.error("Unexpected exception when choosing server using zone aware
logic", e);
        }
        if (server != null) {
            return server;
        } else {
            logger.debug("Zone avoidance logic is not invoked.");
            return super.chooseServer(key);
        }
    }
```

从源码中我们可以看到，只有当负载均衡器中维护的实例所属的 Zone 区域的个数大于 1 的时候才会执行这里的选择策略，否则还是将使用父类的实现。当 Zone 区域的个数大于 1 的时候，它的实现步骤如下所示。

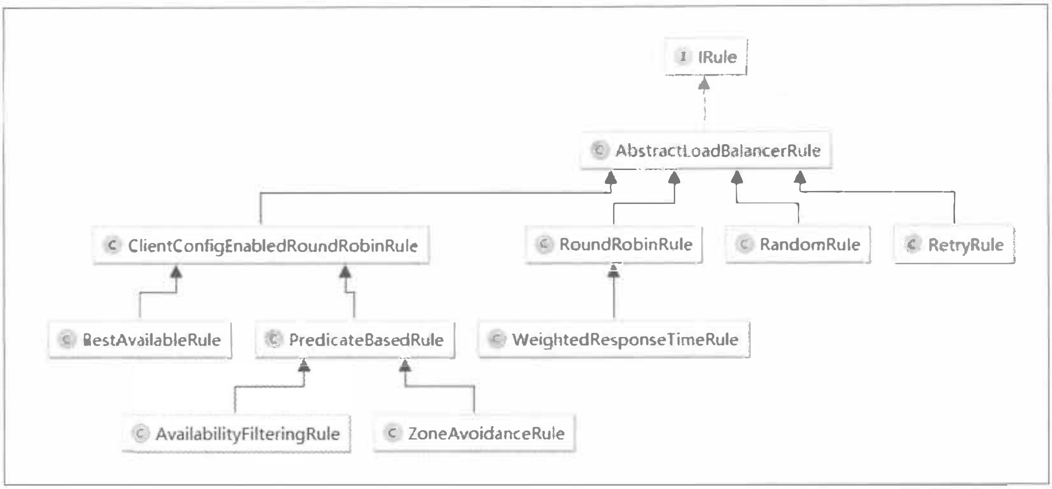
- 调用 ZoneAvoidanceRule 中的静态方法 createSnapshot(lbStats)，为当前负载均衡器中所有的 Zone 区域分别创建快照，保存在 Map zoneSnapshot 中，这些快照中的数据将用于后续的算法。
- 调用 ZoneAvoidanceRule 中的静态方法 getAvailableZones(zoneSnapshot, triggeringLoad.get(), triggeringBlackoutPercentage.get())，来获取可用的 Zone 区域集合，在该函数中会通过 Zone 区域快照中的统计数据来实现可用区的挑选。
    - 首先它会剔除符合这些规则的 Zone 区域：所属实例数为零的 Zone 区域；Zone 区域内实例的平均负载小于零，或者实例故障率（断路器断开次数/实例数）大于等于阈值（默认为 0.99999）。
    - 然后根据 Zone 区域的实例平均负载计算出最差的 Zone 区域，这里的最差指的是实例平均负载最高的 Zone 区域。
    - 如果在上面的过程中没有符合剔除要求的区域，同时实例最大平均负载小于阈值（默认为 20%），就直接返回所有 Zone 区域为可用区域。否则，从最坏 Zone 区域集合中随机选择一个，将它从可用 Zone 区域集合中剔除。
- 当获得的可用 Zone 区域集合不为空，并且个数小于 Zone 区域总数，就随机选择一个 Zone 区域。
- 在确定了某个 Zone 区域后，则获取了对应 Zone 区域的服务均衡器，并调用 chooseServer 来选择具体的服务实例，而在 chooseServer 中将使用 IRule 接口的 choose 函数来选择具体的服务实例。在这里，IRule 接口的实现会使用 ZoneAvoidanceRule 来挑选出具体的服务实例。

关于 ZoneAvoidanceRule 的策略以及其他一些还未提到的负载均衡策略，我们将在下一节做更加详细的解读。

负载均衡策略

通过上面的源码解读，我们已经对 Ribbon 实现的负载均衡器以及其中包含的服务实例过滤器、服务实例信息的存储对象、区域的信息快照等都有了深入的认识和理解，但是对于负载均衡器中的服务实例选择策略只是讲解了几个默认实现的内容，而对于 IRule 的其他实现还没有详细解读，下面我们来看看在 Ribbon 中都提供了哪些负载均衡的策略实现。

如下图所示，可以看到在 Ribbon 中实现了非常多的选择策略，其中也包含了我们在前面内容中提到过的 RoundRobinRule 和 ZoneAvoidanceRule。下面我们来详细解读一下 IRule 接口的各个实现。



AbstractLoadBalancerRule

负载均衡策略的抽象类，在该抽象类中定义了负载均衡器 ILoadBalancer 对象，该对象能够在具体实现选择服务策略时，获取到一些负载均衡器中维护的信息来作为分配依据，并以此设计一些算法来实现针对特定场景的高效策略。

```
public abstract class AbstractLoadBalancerRule implements IRule, IClientConfigAware {

    private ILoadBalancer lb;

    @Override
    public void setLoadBalancer(ILoadBalancer lb){
        this.lb = lb;
    }

    @Override
    public ILoadBalancer getLoadBalancer(){
        return lb;
    }
}
```

RandomRule

该策略实现了从服务实例清单中随机选择一个服务实例的功能。它的具体实现如下，可以看到 IRule 接口的 choose(Object key) 函数实现，委托给了该类中的 choose(ILoadBalancer lb, Object key)，该方法增加了一个负载均衡器对象的参数。从具体的实现上看，它会使用传入的负载均衡器来获得可用实例列表 upList 和所有实例列表 allList，并通过 rand.nextInt(serverCount) 函数来获取一个随机数，

并将该随机数作为 upList 的索引值来返回具体实例。同时，具体的选择逻辑在一个 while (server == null)循环之内，而根据选择逻辑的实现，正常情况下每次选择都应该选出一个服务实例，如果出现死循环获取不到服务实例时，则很有可能存在并发的 Bug。

```
@Override
public Server choose(Object key) {
    return choose(getLoadBalancer(), key);
}

public Server choose(ILoadBalancer lb, Object key) {
    ...
    Server server = null;
    while (server == null) {
        if (Thread.interrupted()) {
            return null;
        }
        List<Server> upList = lb.getReachableServers();
        List<Server> allList = lb.getAllServers();
        int serverCount = allList.size();
        if (serverCount == 0) {
            return null;
        }
        int index = rand.nextInt(serverCount);
        server = upList.get(index);
        if (server == null) {
            Thread.yield();
            continue;
        }
        if (server.isAlive()) {
            return (server);
        }
        server = null;
        Thread.yield();
    }
    return server;
}
```

RoundRobinRule

该策略实现了按照线性轮询的方式依次选择每个服务实例的功能。它的具体实现如下，其详细结构与 RandomRule 非常类似。除了循环条件不同外，就是从可用列表中获取所谓的逻辑不同。从循环条件中，我们可以看到增加了一个 count 计数变量，该变量会在每次循环之后累加，也就是说，如果一直选择不到 server 超过 10 次，那么就会结束尝试，并打印一个警告信息 No available alive servers after 10 tries from load balancer: ...。

而线性轮询的实现则是通过 AtomicInteger nextServerCyclicCounter 对象实现，每次进行实例选择时通过调用 incrementAndGetModulo 函数实现递增。

```
public Server choose(ILoadBalancer lb, Object key) {
    ...
    Server server = null;
    int count = 0;
    while (server == null && count++ < 10) {
        List<Server> reachableServers = lb.getReachableServers();
        List<Server> allServers = lb.getAllServers();
        int upCount = reachableServers.size();
        int serverCount = allServers.size();
        if ((upCount == 0) || (serverCount == 0)) {
            log.warn("No up servers available from load balancer: " + lb);
            return null;
        }
        int nextServerIndex = incrementAndGetModulo(serverCount);
        server = allServers.get(nextServerIndex);
        if (server == null) {
            Thread.yield();
            continue;
        }
        if (server.isAlive() && (server.isReadyToServe())) {
            return (server);
        }
        server = null;
    }
    if (count >= 10) {
        log.warn("No available alive servers after 10 tries from load balancer: "
                + lb);
    }
    return server;
}
```

RetryRule

该策略实现了一个具备重试机制的实例选择功能。从下面的实现中我们可以看到，在其内部还定义了一个 IRule 对象，默认使用了 RoundRobinRule 实例。而在 choose 方法中则实现了对内部定义的策略进行反复尝试的策略，若期间能够选择到具体的服务实例就返回，若选择不到就根据设置的尝试结束时间为阈值（maxRetryMillis 参数定义的值 + choose 方法开始执行的时间戳），当超过该阈值后就返回 null。

```
public class RetryRule extends AbstractLoadBalancerRule {
    IRule subRule = new RoundRobinRule();
```

```
    long maxRetryMillis = 500;
    ...
    public Server choose(ILoadBalancer lb, Object key) {
        long requestTime = System.currentTimeMillis();
        long deadline = requestTime + maxRetryMillis;
        Server answer = null;
        answer = subRule.choose(key);
        if (((answer == null) || (!answer.isAlive()))
                && (System.currentTimeMillis() < deadline)) {
            InterruptTask task = new InterruptTask(deadline
                    - System.currentTimeMillis());
            while (!Thread.interrupted()) {
                answer = subRule.choose(key);
                if (((answer == null) || (!answer.isAlive()))
                        && (System.currentTimeMillis() < deadline)) {
                    Thread.yield();
                } else {
                    break;
                }
            }
            task.cancel();
        }
        if ((answer == null) || (!answer.isAlive())) {
            return null;
        } else {
            return answer;
        }
    }
    ...
}
```

WeightedResponseTimeRule

该策略是对 `RoundRobinRule` 的扩展，增加了根据实例的运行情况来计算权重，并根据权重来挑选实例，以达到更优的分配效果，它的实现主要有三个核心内容。

定时任务

`WeightedResponseTimeRule` 策略在初始化的时候会通过 `serverWeightTimer.schedule(new DynamicServerWeightTask(), 0, serverWeightTaskTimerInterval)` 启动一个定时任务，用来为每个服务实例计算权重，该任务默认 30 秒执行一次。

```
class DynamicServerWeightTask extends TimerTask {
    public void run() {
        ServerWeight serverWeight = new ServerWeight();
```

```
            try {
                serverWeight.maintainWeights();
            } catch (Throwable t) {
                logger.error("Throwable caught while running DynamicServerWeightTask for
" + name, t);
            }
        }
    }
```

权重计算

在源码中我们可以轻松找到用于存储权重的对象 `List<Double> accumulatedWeights = new ArrayList<Double>()`，该 List 中每个权重值所处的位置对应了负载均衡器维护的服务实例清单中所有实例在清单中的位置。

维护实例权重的计算过程通过 `maintainWeights` 函数实现，具体如下面的代码所示：

```
    public void maintainWeights() {
        ILoadBalancer lb = getLoadBalancer();
        ...
        try {
            logger.info("Weight adjusting job started");
            AbstractLoadBalancer nlb = (AbstractLoadBalancer) lb;
            LoadBalancerStats stats = nlb.getLoadBalancerStats();
            ...
            // 计算所有实例的平均响应时间的总和：totalResponseTime
            double totalResponseTime = 0;
            for (Server server : nlb.getAllServers()) {
                // 如果服务实例的状态快照不在缓存中，那么这里会进行自动加载
                ServerStats ss = stats.getSingleServerStat(server);
                totalResponseTime += ss.getResponseTimeAvg();
            }
            // 逐个计算每个实例的权重：weightSoFar + totalResponseTime - 实例的平均响应时间
            Double weightSoFar = 0.0;
            List<Double> finalWeights = new ArrayList<Double>();
            for (Server server : nlb.getAllServers()) {
                ServerStats ss = stats.getSingleServerStat(server);
                double weight = totalResponseTime - ss.getResponseTimeAvg();
                weightSoFar += weight;
                finalWeights.add(weightSoFar);
            }
            setWeights(finalWeights);
        } catch (Throwable t) {
            logger.error("Exception while dynamically calculating server weights", t);
        } finally {
```

```
        serverWeightAssignmentInProgress.set(false);
    }
}
```

该函数的实现主要分为两个步骤：

- 根据 LoadBalancerStats 中记录的每个实例的统计信息，累加所有实例的平均响应时间，得到总平均响应时间 totalResponseTime，该值会用于后续的计算。
- 为负载均衡器中维护的实例清单逐个计算权重（从第一个开始），计算规则为 weightSoFar+totalResponseTime－实例的平均响应时间，其中 weightSoFar 初始化为零，并且每计算好一个权重需要累加到 weightSoFar 上供下一次计算使用。

举个简单的例子来理解这个计算过程，假设有 4 个实例 A、B、C、D，它们的平均响应时间为 10、40、80、100，所以总响应时间是 10 + 40 + 80 + 100 = 230，每个实例的权重为总响应时间与实例自身的平均响应时间的差的累积所得，所以实例 A、B、C、D 的权重分别如下所示。

- 实例 A：230 − 10 = 220
- 实例 B：220 +（230 − 40）= 410
- 实例 C：410 +（230 − 80）= 560
- 实例 D：560 +（230 − 100）= 690

需要注意的是，这里的权重值只是表示了各实例权重区间的上限，并非某个实例的优先级，所以不是数值越大被选中的概率就越大。那么什么是权重区间呢？以上面例子的计算结果为例，它实际上是为这 4 个实例构建了 4 个不同的区间，每个实例的区间下限是上一个实例的区间上限，而每个实例的区间上限则是我们上面计算并存储于 List accumulatedWeights 中的权重值，其中第一个实例的下限默认为零。所以，根据上面示例的权重计算结果，我们可以得到每个实例的权重区间。

- 实例 A：[0, 220]
- 实例 B：(220, 410]
- 实例 C：(410,560]
- 实例 D：(560，690)

不难发现，实际上每个区间的宽度就是：总的平均响应时间 − 实例的平均响应时间，所以实例的平均响应时间越短、权重区间的宽度越大，而权重区间的宽度越大被选中的概率就越高。可能很多读者会问，这些区间边界的开闭是如何确定的呢？为什么不那么规则？下面我们会通过实例选择算法的解读来解释。

实例选择

WeightedResponseTimeRule 选择实例的实现与之前介绍的算法结构类似，下面是它主体的算法（省略了循环体和一些判断等处理）：

```
public Server choose(ILoadBalancer lb, Object key) {
    ...
        List<Double> currentWeights = accumulatedWeights;
        ...
        List<Server> allList = lb.getAllServers();
        int serverCount = allList.size();
        if (serverCount == 0) {
            return null;
        }
        int serverIndex = 0;
        // 获取最后一个实例的权重
        double maxTotalWeight = currentWeights.size() == 0 ? 0 : currentWeights.get
(currentWeights.size() - 1);
        if (maxTotalWeight < 0.001d) {
            // 如果最后一个实例的权重值小于0.001，则采用父类实现的线性轮询的策略
            server =  super.choose(getLoadBalancer(), key);
            if(server == null) {
                return server;
            }
        } else {
            // 如果最后一个实例的权重值大于等于0.001，就产生一个[0, maxTotalWeight)的随机数
            double randomWeight = random.nextDouble() * maxTotalWeight;
            int n = 0;
            for (Double d : currentWeights) {
            // 遍历维护的权重清单，若权重大于等于随机得到的数值，就选择这个实例
                if (d >= randomWeight) {
                    serverIndex = n;
                    break;
                } else {
                    n++;
                }
            }
            server = allList.get(serverIndex);
        }
    ...
    return server;
}
```

从源码中我们可以看到，选择实例的核心过程就两步：

- 生成一个[0, 最大权重值)区间内的随机数。
- 遍历权重列表，比较权重值与随机数的大小，如果权重值大于等于随机数，就拿当前权重列表的索引值去服务实例列表中获取具体的实例。这就是在上一节中提到的服务实例会根据权重区间挑选的原理，而权重区间边界的开闭原则根据算法，正常每个区间为(x, y]的形式，但是第一个实例和最后一个实例为什么不同呢？由于随机数的最小取值可以为 0，所以第一个实例的下限是闭区间，同时随机数的最大值取不到最大权重值，所以最后一个实例的上限是开区间。

若继续以上面的数据为例进行服务实例的选择，则该方法会从[0, 690)区间中选出一个随机数，比如选出的随机数为 230，由于该值位于第二个区间，所以此时就会选择实例 B 来进行请求。

ClientConfigEnabledRoundRobinRule

该策略较为特殊，我们一般不直接使用它。因为它本身并没有实现什么特殊的处理逻辑，正如下面的源码所示，在它的内部定义了一个 RoundRobinRule 策略，而 choose 函数的实现也正是使用了 RoundRobinRule 的线性轮询机制，所以它实现的功能实际上与 RoundRobinRule 相同，那么定义它有什么特殊的用处呢？

虽然我们不会直接使用该策略，但是通过继承该策略，默认的 choose 就实现了线性轮询机制，在子类中做一些高级策略时通常有可能会存在一些无法实施的情况，那么就可以用父类的实现作为备选。在后文中我们将继续介绍的高级策略均是基于 ClientConfigEnabledRoundRobinRule 的扩展。

```
public class ClientConfigEnabledRoundRobinRule extends AbstractLoadBalancerRule {

    RoundRobinRule roundRobinRule = new RoundRobinRule();
    ...
    @Override
    public Server choose(Object key) {
        if (roundRobinRule != null) {
            return roundRobinRule.choose(key);
        } else {
            throw new IllegalArgumentException(
                    "This class has not been initialized with the RoundRobinRule class");
        }
    }
}
```

BestAvailableRule

该策略继承自 ClientConfigEnabledRoundRobinRule，在实现中它注入了负载

均衡器的统计对象 LoadBalancerStats，同时在具体的 choose 算法中利用 LoadBalancerStats 保存的实例统计信息来选择满足要求的实例。从如下源码中我们可以看到，它通过遍历负载均衡器中维护的所有服务实例，会过滤掉故障的实例，并找出并发请求数最小的一个，所以该策略的特性是可选出最空闲的实例。

```
public Server choose(Object key) {
    if (loadBalancerStats == null) {
        return super.choose(key);
    }
    List<Server> serverList = getLoadBalancer().getAllServers();
    int minimalConcurrentConnections = Integer.MAX_VALUE;
    long currentTime = System.currentTimeMillis();
    Server chosen = null;
    for (Server server: serverList) {
        ServerStats serverStats = loadBalancerStats.getSingleServerStat(server);
        if (!serverStats.isCircuitBreakerTripped(currentTime)) {
            int concurrentConnections = serverStats.getActiveRequestsCount(currentTime);
            if (concurrentConnections < minimalConcurrentConnections) {
                minimalConcurrentConnections = concurrentConnections;
                chosen = server;
            }
        }
    }
    if (chosen == null) {
        return super.choose(key);
    } else {
        return chosen;
    }
}
```

同时，由于该算法的核心依据是统计对象 loadBalancerStats，当其为空的时候，该策略是无法执行的。所以从源码中我们可以看到，当 loadBalancerStats 为空的时候，它会采用父类的线性轮询策略，正如我们在介绍 ClientConfigEnabledRoundRobinRule 时那样，它的子类在无法满足实现高级策略的时候，可以使用线性轮询策略的特性。后面将要介绍的策略因为也都继承自 ClientConfigEnabledRoundRobinRule，所以它们都会具有这样的特性。

PredicateBasedRule

这是一个抽象策略，它也继承了 ClientConfigEnabledRoundRobinRule，从其命名中可以猜出这是一个基于 Predicate 实现的策略，Predicate 是 Google Guava Collection 工具对集合进行过滤的条件接口。

如下面的源码所示，它定义了一个抽象函数 getPredicate 来获取 AbstractServerPredicate 对象的实现，而在 choose 函数中，通过 AbstractServerPredicate 的 chooseRoundRobinAfterFiltering 函数来选出具体的服务实例。从该函数的命名我们也大致能猜出它的基础逻辑：先通过子类中实现的 Predicate 逻辑来过滤一部分服务实例，然后再以线性轮询的方式从过滤后的实例清单中选出一个。

```
public abstract class PredicateBasedRule extends ClientConfigEnabledRoundRobinRule {

    public abstract AbstractServerPredicate getPredicate();

    @Override
    public Server choose(Object key) {
        ILoadBalancer lb = getLoadBalancer();
        Optional<Server> server = getPredicate().chooseRoundRobinAfterFiltering
(lb.getAllServers(), key);
        if (server.isPresent()) {
            return server.get();
        } else {
            return null;
        }
    }
}
```

通过下面 AbstractServerPredicate 的源码片段，可以证实我们上面所做的猜测。在上面 choose 函数中调用的 chooseRoundRobinAfterFiltering 方法先通过内部定义的 getEligibleServers 函数来获取备选的实例清单（实现了过滤），如果返回的清单为空，则用 Optional.absent() 来表示不存在，反之则以线性轮询的方式从备选清单中获取一个实例。

```
public abstract class AbstractServerPredicate implements Predicate<PredicateKey> {

    ...

    public Optional<Server> chooseRoundRobinAfterFiltering(List<Server> servers,
Object loadBalancerKey) {
        List<Server> eligible = getEligibleServers(servers, loadBalancerKey);
        if (eligible.size() == 0) {
            return Optional.absent();
        }
        return Optional.of(eligible.get(nextIndex.getAndIncrement() % eligible.
size()));
    }

    public List<Server> getEligibleServers(List<Server> servers, Object
loadBalancerKey) {
```

```
            if (loadBalancerKey == null) {
                return ImmutableList.copyOf(Iterables.filter(servers, this.getServer-
OnlyPredicate()));
            } else {
                List<Server> results = Lists.newArrayList();
                for (Server server: servers) {
                    if (this.apply(new PredicateKey(loadBalancerKey, server))) {
                        results.add(server);
                    }
                }
                return results;
            }
        }
    }
```

在了解了整体逻辑之后，我们来详细看看实现过滤功能的 getEligibleServers 函数。从源码上看，它的实现结构简单清晰，通过遍历服务清单，使用 this.apply 方法来判断实例是否需要保留，如果是就添加到结果列表中。

可能到这里，不熟悉 Google Guava Collections 集合工具的读者会感到困惑，这个 apply 在 AbstractServerPredicate 中找不到它的定义，那么它是如何实现过滤的呢？实际上，AbstractServerPredicate 实现了 com.google.common.base.Predicate 接口，而 apply 方法是该接口中的定义，主要用来实现过滤条件的判断逻辑，它输入的参数则是过滤条件需要用到的一些信息（比如源码中的 new PredicateKey(loadBalancerKey, server)），它传入了关于实例的统计信息和负载均衡器的选择算法传递过来的 key）。既然在 AbstractServerPredicate 中我们未能找到 apply 的实现，所以这里的 chooseRoundRobinAfterFiltering 函数只是定义了一个模板策略："先过滤清单，再轮询选择"。对于如何过滤，需要我们在 AbstractServerPredicate 的子类中实现 apply 方法来确定具体的过滤策略。

后面我们将要介绍的两个策略就是基于此抽象策略实现，只是它们使用了不同的 Predicate 实现来完成过滤逻辑以达到不同的实例选择效果。

> Google Guava Collections 是一个对 Java Collections Framework 增强和扩展的开源项目。虽然 Java Collections Framework 已经能够满足我们大多数情况下使用集合的要求，但是当遇到一些特殊的情况时我们的代码会比较冗长且容易出错。Guava Collections 可以帮助我们让集合操作代码更为简短精练并大大增强代码的可读性。

AvailabilityFilteringRule

该策略继承自上面介绍的抽象策略 PredicateBasedRule，所以它也继承了"先过

滤清单，再轮询选择”的基本处理逻辑，其中过滤条件使用了 AvailabilityPredicate：

```
public class AvailabilityPredicate extends  AbstractServerPredicate {

    ...

    public boolean apply(@Nullable PredicateKey input) {
        LoadBalancerStats stats = getLBStats();
        if (stats == null) {
            return true;
        }
        return !shouldSkipServer(stats.getSingleServerStat(input.getServer()));
    }

    private boolean shouldSkipServer(ServerStats stats) {
        if ((CIRCUIT_BREAKER_FILTERING.get() && stats.isCircuitBreakerTripped())
                || stats.getActiveRequestsCount() >= activeConnectionsLimit.get()) {
            return true;
        }
        return false;
    }
}
```

从上述源码中，我们可以知道它的主要过滤逻辑位于 shouldSkipServer 方法中，它主要判断服务实例的两项内容：

- 是否故障，即断路器是否生效已断开。
- 实例的并发请求数大于阈值，默认值为 $2^{32}-1$，该配置可通过参数<clientName>.<nameSpace>.ActiveConnectionsLimit 来修改。

这两项内容中只要有一个满足 apply 就返回 false（代表该节点可能存在故障或负载过高），都不满足就返回 true。

在该策略中，除了实现了上面的过滤方法之外，对于 choose 的策略也做了一些改进优化，所以父类的实现对于它来说只是一个备用选项，其具体实现如下所示：

```
public Server choose(Object key) {
    int count = 0;
    Server server = roundRobinRule.choose(key);
    while (count++ <= 10) {
        if (predicate.apply(new PredicateKey(server))) {
            return server;
        }
        server = roundRobinRule.choose(key);
    }
    return super.choose(key);
}
```

可以看到，它并没有像在父类中那样，先遍历所有的节点进行过滤，然后在过滤后的集合中选择实例。而是先以线性的方式选择一个实例，接着用过滤条件来判断该实例是否满足要求，若满足就直接使用该实例，若不满足要求就再选择下一个实例，并检查是否满足要求，如此循环进行，当这个过程重复了 10 次还是没有找到符合要求的实例，就采用父类的实现方案。

简单地说，该策略通过线性抽样的方式直接尝试寻找可用且较空闲的实例来使用，优化了父类每次都要遍历所有实例的开销。

ZoneAvoidanceRule

该策略我们在介绍负载均衡器 ZoneAwareLoadBalancer 时已经提到过，它也是 PredicateBasedRule 的具体实现类。在之前的介绍中主要针对 ZoneAvoidanceRule 中用于选择 Zone 区域策略的一些静态函数，比如 createSnapshot、getAvailableZones。在这里我们将详细看看 ZoneAvoidanceRule 作为服务实例过滤条件的实现原理。从下面 ZoneAvoidanceRule 的源码片段中可以看到，它使用了 CompositePredicate 来进行服务实例清单的过滤。这是一个组合过滤条件，在其构造函数中，它以 ZoneAvoidancePredicate 为主过滤条件，AvailabilityPredicate 为次过滤条件初始化了组合过滤条件的实例。

```
public class ZoneAvoidanceRule extends PredicateBasedRule {

    ...
    private CompositePredicate compositePredicate;

    public ZoneAvoidanceRule() {
        super();
        ZoneAvoidancePredicate zonePredicate = new ZoneAvoidancePredicate(this);
        AvailabilityPredicate availabilityPredicate = new AvailabilityPredicate(this);
        compositePredicate = createCompositePredicate(zonePredicate,
availabilityPredicate);
    }
    ...
}
```

ZoneAvoidanceRule 在实现的时候并没有像 AvailabilityFilteringRule 那样重写 choose 函数来优化，所以它完全遵循了父类的过滤主逻辑：“先过滤清单，再轮询选择”。其中过滤清单的条件就是我们上面提到的以 ZoneAvoidancePredicate 为主过滤条件、AvailabilityPredicate 为次过滤条件的组合过滤条件 Composite-Predicate。从 CompositePredicate 的源码片段中，我们可以看到它定义了一个主过滤条件 AbstractServerPredicate delegate 以及一组次过滤条件列表 List

fallbacks，所以它的次过滤列表是可以拥有多个的，并且由于它采用了 List 存储所以次过滤条件是按顺序执行的。

```
public class CompositePredicate extends AbstractServerPredicate {

    private AbstractServerPredicate delegate;
    private List<AbstractServerPredicate> fallbacks = Lists.newArrayList();

    private int minimalFilteredServers = 1;
    private float minimalFilteredPercentage = 0;

    @Override
    public List<Server> getEligibleServers(List<Server> servers, Object
loadBalancerKey) {
        List<Server> result = super.getEligibleServers(servers, loadBalancerKey);
        Iterator<AbstractServerPredicate> i = fallbacks.iterator();
        while (!(result.size() >= minimalFilteredServers && result.size() > (int)
(servers.size() * minimalFilteredPercentage))
                && i.hasNext()) {
            AbstractServerPredicate predicate = i.next();
            result = predicate.getEligibleServers(servers, loadBalancerKey);
        }
        return result;
    }

}
```

在获取过滤结果的实现函数 getEligibleServers 中，它的处理逻辑如下所示。

- 使用主过滤条件对所有实例过滤并返回过滤后的实例清单。
- 依次使用次过滤条件列表中的过滤条件对主过滤条件的结果进行过滤。
- 每次过滤之后（包括主过滤条件和次过滤条件），都需要判断下面两个条件，只要有一个符合就不再进行过滤，将当前结果返回供线性轮询算法选择：
  - 过滤后的实例总数 >= 最小过滤实例数（minimalFilteredServers，默认为 1）。
  - 过滤后的实例比例 > 最小过滤百分比（minimalFilteredPercentage，默认为 0）。

配置详解

通过上一节对 Spring Cloud Ribbon 的源码分析，相信读者对于 Ribbon 的几个重要接口都已经有所了解。下面，我们将详细介绍 Ribbon 在使用时的各种配置方式。

自动化配置

由于 Ribbon 中定义的每一个接口都有多种不同的策略实现，同时这些接口之间又有一定的依赖关系，这使得第一次使用 Ribbon 的开发者很难上手，不知道如何选择具体的实现策略以及如何组织它们的关系。Spring Cloud Ribbon 中的自动化配置恰恰能够解决这样的痛点，在引入 Spring Cloud Ribbon 的依赖之后，就能够自动化构建下面这些接口的实现。

- IClientConfig：Ribbon 的客户端配置，默认采用 com.netflix.client.config.DefaultClientConfigImpl 实现。
- IRule：Ribbon 的负载均衡策略，默认采用 com.netflix.loadbalancer.ZoneAvoidanceRule 实现，该策略能够在多区域环境下选出最佳区域的实例进行访问。
- IPing：Ribbon 的实例检查策略，默认采用 com.netflix.loadbalancer.NoOpPing 实现，该检查策略是一个特殊的实现，实际上它并不会检查实例是否可用，而是始终返回 true，默认认为所有服务实例都是可用的。
- ServerList<Server>：服务实例清单的维护机制，默认采用 com.netflix.loadbalancer.ConfigurationBasedServerList 实现。
- ServerListFilter<Server>：服务实例清单过滤机制，默认采用 org.springframework.cloud.netflix.ribbon.ZonePreferenceServerListFilter 实现，该策略能够优先过滤出与请求调用方处于同区域的服务实例。
- ILoadBalancer：负载均衡器，默认采用 com.netflix.loadbalancer.ZoneAwareLoadBalancer 实现，它具备了区域感知的能力。

上面这些自动化配置内容仅在没有引入 Spring Cloud Eureka 等服务治理框架时如此，在同时引入 Eureka 和 Ribbon 依赖时，自动化配置会有一些不同，后续我们会做详细的介绍。

通过自动化配置的实现，我们可以轻松地实现客户端负载均衡。同时，针对一些个性化需求，我们也可以方便地替换上面的这些默认实现。只需在 Spring Boot 应用中创建对应的实现实例就能覆盖这些默认的配置实现。比如下面的配置内容，由于创建了 PingUrl 实例，所以默认的 NoOpPing 就不会被创建。

```
@Configuration
public class MyRibbonConfiguration {
    @Bean
    public IPing ribbonPing(IClientConfig config) {
        return new PingUrl();
    }
}
```

另外，也可以通过使用@RibbonClient 注解来实现更细粒度的客户端配置，比如下面的代码实现了为 hello-service 服务使用 HelloServiceConfiguration 中的配置。

```
@Configuration
@RibbonClient(name = "hello-service", configuration = HelloServiceConfiguration.
class)
public class RibbonConfiguration {
}
```

Camden 版本对 RibbonClient 配置的优化

上面我们介绍了在 Brixton 版本中对 RibbonClient 的 IPing、IRule 等接口实现进行个性化定制的方法，主要通过独立创建一个 Configuration 类来定义 IPing、IRule 等接口的具体实现 Bean，然后在创建 RibbonClient 时指定要使用的具体 Configuration 类来覆盖自动化配置的默认实现。虽然这种方式已经能够实现个性化的定义，但是当有大量这类配置的时候，对于各个 RibbonClient 的指定配置信息都将分散在这些配置类的注解定义中，这使得管理和修改都变得非常不方便。所以，在 Camden 版本中，Spring Cloud Ribbon 对 RibbonClient 定义个性化配置的方法做了进一步优化。可以直接通过 <clientName>.ribbon.<key>=<value>的形式进行配置。比如我们要实现与上面例子一样的配置（将 hello-service 服务客户端的 IPing 接口实现替换为 PingUrl），只需在 application.properties 配置中增加下面的内容即可：

```
hello-service.ribbon.NFLoadBalancerPingClassName=com.netflix.loadbalancer.PingUrl
```

其中 hello-service 为服务名，NFLoadBalancerPingClassName 参数用来指定具体的 IPing 接口实现类。在 Camden 版本中，Spring Cloud Ribbon 新增了一个 org.springframework.cloud.netflix.ribbon.PropertiesFactory 类来动态地为 RibbonClient 创建这些接口实现。

```
public class PropertiesFactory {
    @Autowired
    private Environment environment;

    private Map<Class, String> classToProperty = new HashMap<>();

    public PropertiesFactory() {
        classToProperty.put(ILoadBalancer.class, "NFLoadBalancerClassName");
        classToProperty.put(IPing.class, "NFLoadBalancerPingClassName");
        classToProperty.put(IRule.class, "NFLoadBalancerRuleClassName");
        classToProperty.put(ServerList.class, "NIWSServerListClassName");
        classToProperty.put(ServerListFilter.class, "NIWSServerListFilterClassName");
    }
```

```
    public boolean isSet(Class clazz, String name) {
        return StringUtils.hasText(getClassName(clazz, name));
    }

    public String getClassName(Class clazz, String name) {
        if (this.classToProperty.containsKey(clazz)) {
            String classNameProperty = this.classToProperty.get(clazz);
            String className = environment.getProperty(name + "." + NAMESPACE + "."
+ classNameProperty);
            return className;
        }
        return null;
    }

    @SuppressWarnings("unchecked")
    public <C> C get(Class<C> clazz, IClientConfig config, String name) {
        String className = getClassName(clazz, name);
        if (StringUtils.hasText(className)) {
            try {
                Class<?> toInstantiate = Class.forName(className);
                return (C) instantiateWithConfig(toInstantiate, config);
            } catch (ClassNotFoundException e) {
                throw new IllegalArgumentException("Unknown class to load
"+className+" for class " + clazz + " named " + name);
            }
        }
        return null;
    }
}
```

从上述源码定义中可以看到，除了 NFLoadBalancerPingClassName 参数之外，还提供了其他几个接口的动态配置实现，具体如下所述。

- NFLoadBalancerClassName：配置 ILoadBalancer 接口的实现。
- NFLoadBalancerPingClassName：配置 IPing 接口的实现。
- NFLoadBalancerRuleClassName：配置 IRule 接口的实现。
- NIWSServerListClassName：配置 ServerList 接口的实现。
- NIWSServerListFilterClassName：配置 ServerListFilter 接口的实现。

所以，在 Camden 版本中我们可以通过配置的方式，更加方便地为 RibbonClient 指定 ILoadBalancer、IPing、IRule、ServerList 和 ServerListFilter 的定制化实现。

参数配置

对于 Ribbon 的参数配置通常有两种方式：全局配置以及指定客户端配置。

- 全局配置的方式很简单，只需使用 ribbon.<key>=<value>格式进行配置即可。其中，<key>代表了 Ribbon 客户端配置的参数名，<value>则代表了对应参数的值。比如，我们可以像下面这样全局配置 Ribbon 创建连接的超时时间：

```
ribbon.ConnectTimeout=250
```

  全局配置可以作为默认值进行设置，当指定客户端配置了相应 key 的值时，将覆盖全局配置的内容。

- 指定客户端的配置方式采用<client>.ribbon.<key>=<value>的格式进行配置。其中，<key>和<value>的含义同全局配置相同，而<client>代表了客户端的名称，如上文中我们在@RibbonClient 中指定的名称，也可以将它理解为是一个服务名。为了方便理解这种配置方式，我们举一个具体的例子：假设，有一个服务消费者通过 RestTemplate 来访问 hello-service 服务的/hello 接口，这时我们会这样调用 restTemplate.getForEntity("http://hello-service/hello", String.class).getBody();。如果没有服务治理框架的帮助，我们需要为该客户端指定具体的实例清单，可以指定服务名来做详细的配置，具体如下：

```
hello-service.ribbon.listOfServers=localhost:8001,localhost:8002, localhost:8003
```

对于 Ribbon 参数的 key 以及 value 类型的定义，可以通过查看 com.netflix.client.config.CommonClientConfigKey 类获得更为详细的配置内容，在本书中不进行详细介绍。

与 Eureka 结合

当在 Spring Cloud 的应用中同时引入 Spring Cloud Ribbon 和 Spring Cloud Eureka 依赖时，会触发 Eureka 中实现的对 Ribbon 的自动化配置。这时 ServerList 的维护机制实现将被 com.netflix.niws.loadbalancer.DiscoveryEnabledNIWSServerList 的实例所覆盖，该实现会将服务清单列表交给 Eureka 的服务治理机制来进行维护；IPing 的实现将被 com.netflix.niws.loadbalancer.NIWSDiscoveryPing 的实例所覆盖，该实现也将实例检查的任务交给了服务治理框架来进行维护。默认情况下，用于获取实例请求的 ServerList 接口实现将采用 Spring Cloud Eureka 中封装的 org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList，其目的是为了让实例维护策略更加通用，所以将使用物理元数据来进行负载均衡，而不是使用原生的 AWS AMI 元数据。

在与 Spring Cloud Eureka 结合使用的时候，我们的配置将会变得更加简单。不再需要通过类似 `hello-service.ribbon.listOfServers` 的参数来指定具体的服务实例清单，因为 Eureka 将会为我们维护所有服务的实例清单。而对于 Ribbon 的参数配置，我们依然可以采用之前的两种配置方式来实现，而指定客户端的配置方式可以直接使用 Eureka 中的服务名作为`<client>`来完成针对各个微服务的个性化配置。

此外，由于 Spring Cloud Ribbon 默认实现了区域亲和策略，所以，我们可以通过 Eureka 实例的元数据配置来实现区域化的实例配置方案。比如，可以将处于不同机房的实例配置成不同的区域值，以作为跨区域的容错机制实现。而实现的方式非常简单，只需在服务实例的元数据中增加 `zone` 参数来指定自己所在的区域，比如：

```
eureka.instance.metadataMap.zone=shanghai
```

在 Spring Cloud Ribbon 与 Spring Cloud Eureka 结合的工程中，我们也可以通过参数配置的方式来禁用 Eureka 对 Ribbon 服务实例的维护实现。只需在配置文件中加入如下参数，这时我们对于服务实例的维护就又将回归到使用`<client>.ribbon.listOfServers` 参数配置的方式来实现了。

```
ribbon.eureka.enabled=false
```

重试机制

由于 Spring Cloud Eureka 实现的服务治理机制强调了 CAP 原理中的 AP，即可用性与可靠性，它与 ZooKeeper 这类强调 CP（一致性、可靠性）的服务治理框架最大的区别就是，Eureka 为了实现更高的服务可用性，牺牲了一定的一致性，在极端情况下它宁愿接受故障实例也不要丢掉“健康”实例，比如，当服务注册中心的网络发生故障断开时，由于所有的服务实例无法维持续约心跳，在强调 AP 的服务治理中将会把所有服务实例都剔除掉，而 Eureka 则会因为超过 85%的实例丢失心跳而会触发保护机制，注册中心将会保留此时的所有节点，以实现服务间依然可以进行互相调用的场景，即使其中有部分故障节点，但这样做可以继续保障大多数的服务正常消费。

由于 Spring Cloud Eureka 在可用性与一致性上的取舍，不论是由于触发了保护机制还是服务剔除的延迟，引起服务调用到故障实例的时候，我们还是希望能够增强对这类问题的容错。所以，我们在实现服务调用的时候通常会加入一些重试机制。在目前我们使用的 Brixt on版本中，对于重试机制的实现需要我们自己来扩展完成。而从 Camden SR2 版本开始，Spring Cloud 整合了 Spring Retry 来增强 `RestTemplate` 的重试能力，对于开发者来说只需通过简单的配置，原来那些通过 `RestTemplate` 实现的服务访问就会自动根据配置来实现重试策略。

以我们之前对 hello-service 服务的调用为例，可以在配置文件中增加如下内容：

```
spring.cloud.loadbalancer.retry.enabled=true

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=10000

hello-service.ribbon.ConnectTimeout=250
hello-service.ribbon.ReadTimeout=1000
hello-service.ribbon.OkToRetryOnAllOperations=true
hello-service.ribbon.MaxAutoRetriesNextServer=2
hello-service.ribbon.MaxAutoRetries=1
```

其中各参数的配置说明如下所示。

- spring.cloud.loadbalancer.retry.enabled：该参数用来开启重试机制，它默认是关闭的。这里需要注意，官方文档中的配置参数少了 enabled。该参数的源码定义如下：

```
@ConfigurationProperties("spring.cloud.loadbalancer.retry")
public class LoadBalancerRetryProperties {
    private boolean enabled = false;
    ...
}
```

- hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds：断路器的超时时间需要大于 Ribbon 的超时时间，不然不会触发重试。
- hello-service.ribbon.ConnectTimeout：请求连接的超时时间。
- hello-service.ribbon.ReadTimeout：请求处理的超时时间。
- hello-service.ribbon.OkToRetryOnAllOperations：对所有操作请求都进行重试。
- hello-service.ribbon.MaxAutoRetriesNextServer：切换实例的重试次数。
- hello-service.ribbon.MaxAutoRetries：对当前实例的重试次数。

根据如上配置，当访问到故障请求的时候，它会再尝试访问一次当前实例（次数由 MaxAutoRetries 配置），如果不行，就换一个实例进行访问，如果还是不行，再换一次实例访问（更换次数由 MaxAutoRetriesNextServer 配置），如果依然不行，返回失败信息。

第 5 章

服务容错保护：Spring Cloud Hystrix

在微服务架构中，我们将系统拆分成了很多服务单元，各单元的应用间通过服务注册与订阅的方式互相依赖。由于每个单元都在不同的进程中运行，依赖通过远程调用的方式执行，这样就有可能因为网络原因或是依赖服务自身问题出现调用故障或延迟，而这些问题会直接导致调用方的对外服务也出现延迟，若此时调用方的请求不断增加，最后就会因等待出现故障的依赖方响应形成任务积压，最终导致自身服务的瘫痪。

举个例子，在一个电商网站中，我们可能会将系统拆分成用户、订单、库存、积分、评论等一系列服务单元。用户创建一个订单的时候，客户端将调用订单服务的创建订单接口，此时创建订单接口又会向库存服务来请求出货（判断是否有足够库存来出货）。此时若库存服务因自身处理逻辑等原因造成响应缓慢，会直接导致创建订单服务的线程被挂起，以等待库存申请服务的响应，在漫长的等待之后用户会因为请求库存失败而得到创建订单失败的结果。如果在高并发情况之下，因这些挂起的线程在等待库存服务的响应而未能释放，使得后续到来的创建订单请求被阻塞，最终导致订单服务也不可用。

在微服务架构中，存在着那么多的服务单元，若一个单元出现故障，就很容易因依赖关系而引发故障的蔓延，最终导致整个系统的瘫痪，这样的架构相较传统架构更加不稳定。为了解决这样的问题，产生了断路器等一系列的服务保护机制。

断路器模式源于 Martin Fowler 的 *Circuit Breaker* 一文。“断路器”本身是一种开关装置，用于在电路上保护线路过载，当线路中有电器发生短路时，“断路器”能够及时切断故障电路，防止发生过载、发热甚至起火等严重后果。

在分布式架构中，断路器模式的作用也是类似的，当某个服务单元发生故障（类似用

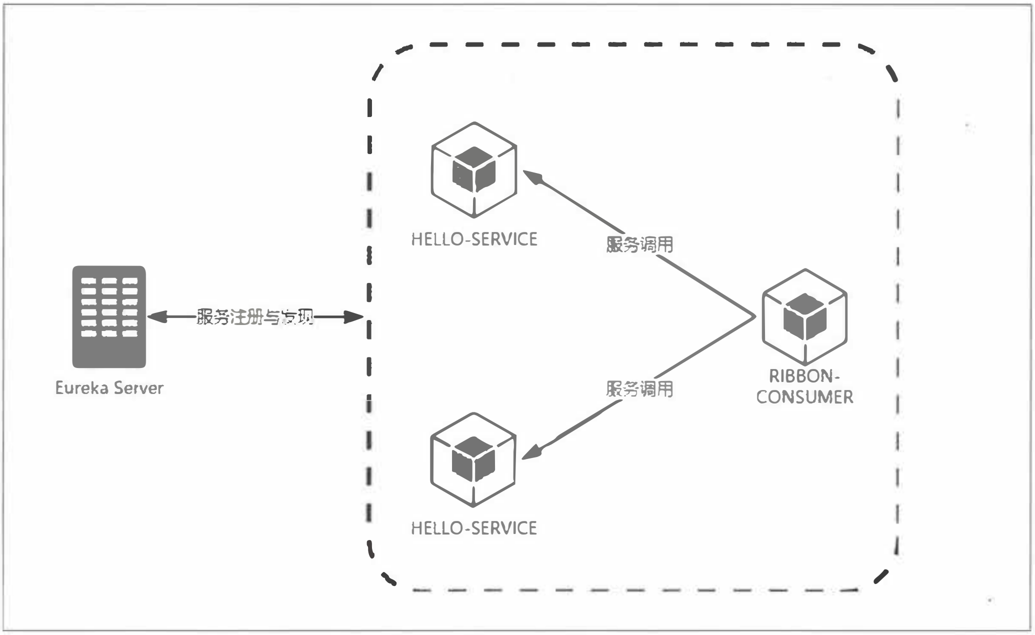
电器发生短路）之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个错误响应，而不是长时间的等待。这样就不会使得线程因调用故障服务被长时间占用不释放，避免了故障在分布式系统中的蔓延。

针对上述问题，Spring Cloud Hystrix 实现了断路器、线程隔离等一系列服务保护功能。它也是基于 Netflix 的开源框架 Hystrix 实现的，该框架的目标在于通过控制那些访问远程系统、服务和第三方库的节点，从而对延迟和故障提供更强大的容错能力。Hystrix 具备服务降级、服务熔断、线程和信号隔离、请求缓存、请求合并以及服务监控等强大功能。

接下来，我们就从一个简单示例开始对 Spring Cloud Hystrix 的学习与使用。

快速入门

在开始使用 Spring Cloud Hystrix 实现断路器之前，我们先用之前实现的一些内容作为基础，构建一个如下图架构所示的服务调用关系。



我们在这里需要启动的工程有如下一些。

- `eureka-server` 工程：服务注册中心，端口为 1111。
- `hello-service` 工程：`HELLO-SERVICE` 的服务单元，两个实例启动端口分别为 8081 和 8082。

- `ribbon-consume` 工程：使用 Ribbon 实现的服务消费者，端口为 9000。

在未加入断路器之前，关闭 8081 的实例，发送 GET 请求到 `http://localhost:9000/ribbon-consumer`，可以获得下面的输出：

```
{
  "timestamp": 1473992080343,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "org.springframework.web.client.ResourceAccessException",
  "message": "I/O error on GET request for \"http://HELLO-SERVICE/hello\":
Connection refused: connect; nested exception is java.net.ConnectException: Connection
refused: connect",
  "path": "/ribbon-consumer"
}
```

下面我们开始引入 Spring Cloud Hystrix。

- 在 `ribbon-consumer` 工程的 `pom.xml` 的 dependency 节点中引入 `spring-cloud-starter-hystrix` 依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
</dependency>
```

- 在 `ribbon-consumer` 工程的主类 `ConsumerApplication` 中使用`@EnableCircuitBreaker`注解开启断路器功能：

```
@EnableCircuitBreaker
@EnableDiscoveryClient
@SpringBootApplication
public class ConsumerApplication {

    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }

}
```

注意：这里还可以使用 Spring Cloud 应用中的`@SpringCloudApplication`注解来修饰应用主类，该注解的具体定义如下所示。可以看到，该注解中包含了上述我们所引用的三个注解，这也意味着一个 Spring Cloud 标准应用应包含服务发现以及断路器。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public @interface SpringCloudApplication {
}
```

- 改造服务消费方式，新增`HelloService`类，注入`RestTemplate`实例。然后，将在`ConsumerController`中对`RestTemplate`的使用迁移到`helloService`函数中，最后，在`helloService`函数上增加`@HystrixCommand`注解来指定回调方法：

```
@Service
public class HelloService {

    @Autowired
    RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "helloFallback")
    public String helloService() {
        return restTemplate.getForEntity("http://HELLO-SERVICE/hello",
String.class).getBody();
    }

    public String helloFallback() {
        return "error";
    }

}
```

- 修改`ConsumerController`类，注入上面实现的`HelloService`实例，并在`helloConsumer`中进行调用：

```
@RestController
public class ConsumerController {
```

```
    @Autowired
    HelloService helloService;

    @RequestMapping(value = "/ribbon-consumer", method = RequestMethod.GET)
    public String helloConsumer() {
        return helloService.helloService();
    }

}
```

下面，我们来验证一下通过断路器实现的服务回调逻辑，重新启动之前关闭的 8081 端口的 Hello-Service，确保此时服务注册中心、两个 Hello-Service 以及 RIBBON-CONSUMER 均已启动，访问 http://localhost:9000/ribbon-consumer 可以轮询两个 HELLO-SERVICE 并返回一些文字信息。此时我们继续断开 8081 的 HELLO-SERVICE，然后访问 http://localhost:9000/ribbon-consumer，当轮询到 8081 服务端时，输出内容为 error，不再是之前的错误内容，Hystrix 的服务回调生效。除了通过断开具体的服务实例来模拟某个节点无法访问的情况之外，我们还可以模拟一下服务阻塞（长时间未响应）的情况。我们对 HELLO-SERVICE 的/hello 接口做一些修改，具体如下：

```
@RequestMapping(value = "/hello", method = RequestMethod.GET)
public String hello() throws Exception {
    ServiceInstance instance = client.getLocalServiceInstance();
    // 让处理线程等待几秒钟
    int sleepTime = new Random().nextInt(3000);
    logger.info("sleepTime:" + sleepTime);
    Thread.sleep(sleepTime);

    logger.info("/hello, host:" + instance.getHost() + ", service_id:" +
instance.getServiceId());
    return "Hello World";
}
```

通过 Thread.sleep()函数可让/hello 接口的处理线程不是马上返回内容，而是在阻塞几秒之后才返回内容。由于 Hystrix 默认超时时间为 2000 毫秒，所以这里采用了 0 至 3000 的随机数以让处理过程有一定概率发生超时来触发断路器。为了更精准地观察断路器的触发，在消费者调用函数中做一些时间记录，具体如下：

```
@HystrixCommand(fallbackMethod = "helloFallback", commandKey = "helloKey")
public String hello() {
    long start = System.currentTimeMillis();

    // 消费服务的逻辑
    ...
```

```
    long end = System.currentTimeMillis();

    logger.info("Spend time : " + (end - start) );
    return result.toString();
}
```

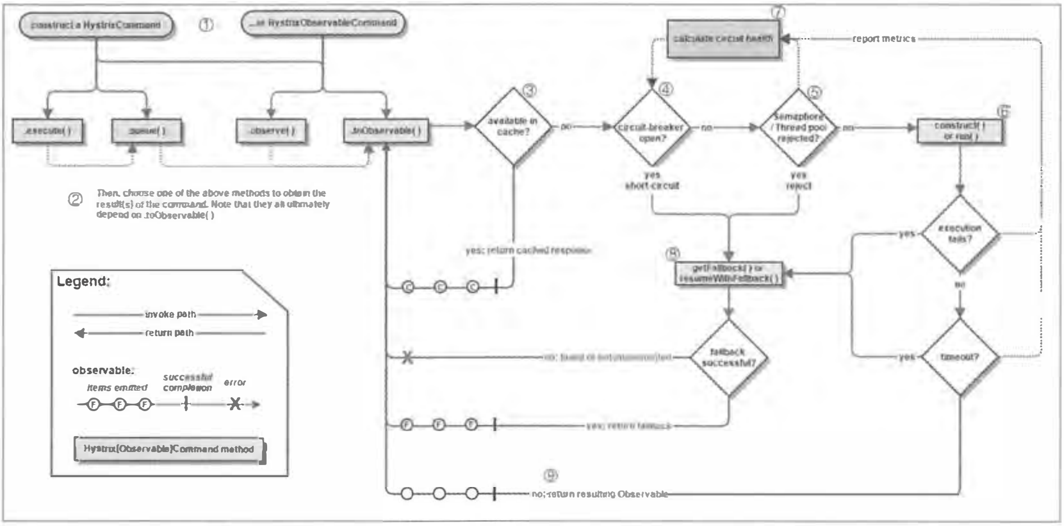
重新启动 HELLO-SERVICE 和 RIBBON-CONSUMER 的实例，连续访问 http://localhost:9000/ribbon-consumer 几次，我们可以观察到，当 RIBBON-CONSUMER 的控制台中输出的 Spend time 大于 2000 的时候，就会返回 error，即服务消费者因调用的服务超时从而触发熔断请求，并调用回调逻辑返回结果。

原理分析

通过上面的快速入门示例，我们对 Hystrix 的使用场景 和使用方法已经有了一个基础的认识。接下来我们通过解读 Netflix Hystrix 官方的流程图来详细了解一下：当一个请求调用了相关服务依赖之后 Hystrix 是如何工作的（即如上例中所示，当访问了 http://localhost:9000/ribbon-consumer 请求之后，在 RIBBON-CONSUMER 中是如何处理的）。

工作流程

下面我们根据图中标记的数字顺序来解释每一个环节的详细内容。



1. 创建 HystrixCommand 或 HystrixObservableCommand 对象

首先，构建一个 HystrixCommand 或是 HystrixObservableCommand 对象，用

来表示对依赖服务的操作请求，同时传递所有需要的参数。从其命名中我们就能知道它采用了“命令模式”来实现对服务调用操作的封装。而这两个 Command 对象分别针对不同的应用场景。

- HystrixCommand：用在依赖的服务返回单个操作结果的时候。
- HystrixObservableCommand：用在依赖的服务返回多个操作结果的时候。

命令模式，将来自客户端的请求封装成一个对象，从而让你可以使用不同的请求对客户端进行参数化。它可以被用于实现“行为请求者”与“行为实现者”的解耦，以便使两者可以适应变化。下面的示例是对命令模式的简单实现：

```
// 接收者
public class Receiver {
    public void action(){
        // 真正的业务逻辑
    }
}
// 抽象命令
interface Command {
    void execute();
}
// 具体命令实现
public class ConcreteCommand implements Command {
    private Receiver receiver;
    public ConcreteCommand(Receiver receiver){
        this.receiver = receiver;
    }
    public void execute() {
        this.receiver.action();
    }
}
// 客户端调用者
public class Invoker {
    private Command command;
  public void setCommand(Command command) {
    this.command = command;
    }
  public void action(){
        this.command.execute();
  }
}
public class Client {
    public static void main(String[] args){
```

```
        Receiver receiver = new Receiver();
        Command command = new ConcreteCommand(receiver);
        Invoker invoker = new Invoker();
        invoker.setCommand(command);
        invoker.action(); //客户端通过调用者来执行命令
    }
}
```

从代码中，我们可以看到这样几个对象。

- Receiver：接收者，它知道如何处理具体的业务逻辑。
- Command：抽象命令，它定义了一个命令对象应具备的一系列命令操作，比如 execute()、undo()、redo()等。当命令操作被调用的时候就会触发接收者去做具体命令对应的业务逻辑。
- ConcreteCommand：具体的命令实现，在这里它绑定了命令操作与接收者之间的关系，execute()命令的实现委托给了 Receiver 的 action()函数。
- Invoker：调用者，它持有一个命令对象，并且可以在需要的时候通过命令对象完成具体的业务逻辑。

从上面的示例中，我们可以看到，调用者 Invoker 与操作者 Receiver 通过 Command 命令接口实现了解耦。对于调用者来说，我们可以为其注入多个命令操作，比如新建文件、复制文件、删除文件这样三个操作，调用者只需在需要的时候直接调用即可，而不需要知道这些操作命令实际是如何实现的。而在这里所提到的 HystrixCommand 和 HystrixObservableCommand 则是在 Hystrix 中对 Command 的进一步抽象定义。在后续的内容中，会逐步展开介绍它的部分内容来帮助理解其运作原理。

从上面的示例中我们也可以发现，Invoker 和 Receiver 的关系非常类似于“请求-响应”模式，所以它比较适用于实现记录日志、撤销操作、队列请求等。

在下面这些情况下应考虑使用命令模式。

- 使用命令模式作为“回调（CallBack）”在面向对象系统中的替代。“CallBack”讲的便是先将一个函数登记上，然后在以后调用此函数。
- 需要在不同的时间指定请求、将请求排队。一个命令对象和原先的请求发出者可以有不同的生命期。换言之，原先的请求发出者可能已经不在了，而命令对象本身仍然是活动的。这时命令的接收者可以是在本地，也可以在网络的另外一个地址。命令对象可以在序列化之后传送到另外一台机器上去。
- 系统需要支持命令的撤销。命令对象可以把状态存储起来，等到客户端需要撤销命令所产生的效果时，可以调用 undo()方法，把命令所产生的效果撤销掉。命令对象还可以提供 redo()方法，以供客户端在需要时再重新实施命令效果。

- 如果要将系统中所有的数据更新到日志里，以便在系统崩溃时，可以根据日志读回所有的数据更新命令，重新调用 Execute() 方法一条一条执行这些命令，从而恢复系统在崩溃前所做的数据更新。

2. 命令执行

从图中我们可以看到一共存在 4 种命令的执行方式，而 Hystrix 在执行时会根据创建的 Command 对象以及具体的情况来选择一个执行。其中 HystrixCommand 实现了下面两个执行方式。

- execute()：同步执行，从依赖的服务返回一个单一的结果对象，或是在发生错误的时候抛出异常。
- queue()：异步执行，直接返回一个 Future 对象，其中包含了服务执行结束时要返回的单一结果对象。

```
R value = command.execute();
Future<R> fValue = command.queue();
```

而 HystrixObservableCommand 实现了另外两种执行方式。

- observe()：返回 Observable 对象，它代表了操作的多个结果，它是一个 Hot Observable。
- toObservable()：同样会返回 Observable 对象，也代表了操作的多个结果，但它返回的是一个 Cold Observable。

```
Observable<R> ohValue = command.observe();
Observable<R> ocValue = command.toObservable();
```

在 Hystrix 的底层实现中大量地使用了 RxJava，为了更容易地理解后续内容，在这里对 RxJava 的观察者-订阅者模式做一个简单的入门介绍。

上面我们所提到的 Observable 对象就是 RxJava 中的核心内容之一，可以把它理解为“事件源”或是“被观察者”，与其对应的 Subscriber 对象，可以理解为“订阅者”或是“观察者”。这两个对象是 RxJava 响应式编程的重要组成部分。

- Observable 用来向订阅者 Subscriber 对象发布事件，Subscriber 对象则在接收到事件后对其进行处理，而在这里所指的事件通常就是对依赖服务的调用。
- 一个 Observable 可以发出多个事件，直到结束或是发生异常。
- Observable 对象每发出一个事件，就会调用对应观察者 Subscriber 对象的 onNext() 方法。
- 每一个 Observable 的执行，最后一定会通过调用 Subscriber.onCompleted() 或者 Subscriber.onError() 来结束该事件的操作流。

下面我们通过一个简单的例子来直观理解一下 Observable 与 Subscribers：

```
// 创建事件源 observable
Observable<String> observable = Observable.create(new Observable.OnSubscribe
<String>() {
    @Override
    public void call(Subscriber<? super String> subscriber) {
        subscriber.onNext("Hello RxJava");
        subscriber.onNext("I am 程序猿 DD");
        subscriber.onCompleted();
    }
});

// 创建订阅者 subscriber
Subscriber<String> subscriber = new Subscriber<String>() {
    @Override
    public void onCompleted() {
    }

    @Override
    public void onError(Throwable e) {
    }
    @Override
    public void onNext(String s) {
        System.out.println("Subscriber : " + s);
    }
};

// 订阅
observable.subscribe(subscriber);
```

在该示例中，创建了一个简单的事件源 observable，一个对事件传递内容输出的订阅者 subscriber，通过 observable.subscribe(subscriber)来触发事件的发布。

在这里我们对于事件源 observable 提到了两个不同的概念：Hot Observable 和 Cold Observable，分别对应了上面 command.observe()和 command.toObservable()的返回对象。其中 Hot Observable，它不论“事件源”是否有“订阅者”，都会在创建后对事件进行发布，所以对于 Hot Observable 的每一个“订阅者”都有可能是从“事件源”的中途开始的，并可能只是看到了整个操作的局部过程。而 Cold Observable 在没有“订阅者”的时候并不会发布事件，而是进行等待，直到有“订阅者”之后才发布事件，所以对于 Cold Observable 的订阅者，它可以保证从一开始看到整个操作的全部过程。

大家从表面上可能会认为只是在 HystrixObservableCommand 中使用了 RxJava，

然而实际上 execute()、queue() 也都使用了 RxJava 来实现。从下面的源码中我们可以看到：

- execute() 是通过 queue() 返回的异步对象 Future<R>的 get() 方法来实现同步执行的。该方法会等待任务执行结束，然后获得 R 类型的结果进行返回。
- queue() 则是通过 toObservable() 来获得一个 Cold Observable，并且通过 toBlocking() 将该 Observable 转换成 BlockingObservable，它可以把数据以阻塞的方式发射出来。而 toFuture 方法则是把 BlockingObservable 转换为一个 Future，该方法只是创建一个 Future 返回，并不会阻塞，这使得消费者可以自己决定如何处理异步操作。而 execute() 就是直接使用了 queue() 返回的 Future 中的阻塞方法 get() 来实现同步操作的。同时通过这种方式转换的 Future 要求 Observable 只发射一个数据，所以这两个实现都只能返回单一结果。

```
public R execute() {
    try {
        return queue().get();
    } catch (Exception e) {
        throw decomposeException(e);
    }
}

public Future<R> queue() {
    final Observable<R> o = toObservable();
    final Future<R> f = o.toBlocking().toFuture();

    if (f.isDone()) {
        // 处理立即抛出的错误
        ...
    }
    return f;
}
```

3. 结果是否被缓存

若当前命令的请求缓存功能是被启用的，并且该命令缓存命中，那么缓存的结果会立即以 Observable 对象的形式返回。

4. 断路器是否打开

在命令结果没有缓存命中的时候，Hystrix 在执行命令前需要检查断路器是否为打开状态：

- 如果断路器是打开的，那么 Hystrix 不会执行命令，而是转接到 fallback 处理逻辑（对应下面第 8 步）。

- 如果断路器是关闭的，那么 Hystrix 跳到第 5 步，检查是否有可用资源来执行命令。

关于断路器的具体实现细节，后续会做更加详细的分析。

5. 线程池/请求队列/信号量是否占满

如果与命令相关的线程池和请求队列，或者信号量（不使用线程池的时候）已经被占满，那么 Hystrix 也不会执行命令，而是转接到 fallback 处理逻辑（对应下面第 8 步）。

需要注意的是，这里 Hystrix 所判断的线程池并非容器的线程池，而是每个依赖服务的专有线程池。Hystrix 为了保证不会因为某个依赖服务的问题影响到其他依赖服务而采用了“舱壁模式”（Bulkhead Pattern）来隔离每个依赖的服务。关于依赖服务的隔离与线程池相关的内容见后续详细介绍。

6. `HystrixObservableCommand.construct()`或 `HystrixCommand.run()`

Hystrix 会根据我们编写的方法来决定采取什么样的方式去请求依赖服务。

- `HystrixCommand.run()`：返回一个单一的结果，或者抛出异常。
- `HystrixObservableCommand.construct()`：返回一个 `Observable` 对象来发射多个结果，或通过 `onError` 发送错误通知。

如果 `run()`或 `construct()`方法的执行时间超过了命令设置的超时阈值，当前处理线程将会抛出一个 `TimeoutException`（如果该命令不在其自身的线程中执行，则会通过单独的计时线程来抛出）。在这种情况下，Hystrix 会转接到 fallback 处理逻辑（第 8 步）。同时，如果当前命令没有被取消或中断，那么它最终会忽略 `run()`或者 `construct()`方法的返回。

如果命令没有抛出异常并返回了结果，那么 Hystrix 在记录一些日志并采集监控报告之后将该结果返回。在使用 `run()`的情况下，Hystrix 会返回一个 `Observable`，它发射单个结果并产生 `onCompleted` 的结束通知；而在使用 `construct()`的情况下，Hystrix 会直接返回该方法产生的 `Observable` 对象。

7. 计算断路器的健康度

Hystrix 会将“成功”、“失败”、“拒绝”、“超时”等信息报告给断路器，而断路器会维护一组计数器来统计这些数据。

断路器会使用这些统计数据来决定是否要将断路器打开，来对某个依赖服务的请求进行“熔断/短路”，直到恢复期结束。若在恢复期结束后，根据统计数据判断如果还是未达到健康指标，就再次“熔断/短路”。

8. fallback 处理

当命令执行失败的时候，Hystrix 会进入 fallback 尝试回退处理，我们通常也称该操作为“服务降级”。而能够引起服务降级处理的情况有下面几种：

- 第 4 步，当前命令处于“熔断/短路”状态，断路器是打开的时候。
- 第 5 步，当前命令的线程池、请求队列或者信号量被占满的时候。
- 第 6 步，`HystrixObservableCommand.construct()` 或 `HystrixCommand.run()` 抛出异常的时候。

在服务降级逻辑中，我们需要实现一个通用的响应结果，并且该结果的处理逻辑应当是从缓存或是根据一些静态逻辑来获取，而不是依赖网络请求获取。如果一定要在降级逻辑中包含网络请求，那么该请求也必须被包装在 `HystrixCommand` 或是 `HystrixObservableCommand` 中，从而形成级联的降级策略，而最终的降级逻辑一定不是一个依赖网络请求的处理，而是一个能够稳定地返回结果的处理逻辑。

在 `HystrixCommand` 和 `HystrixObservableCommand` 中实现降级逻辑时还略有不同：

- 当使用 `HystrixCommand` 的时候，通过实现 `HystrixCommand.getFallback()` 来实现服务降级逻辑。
- 当使用 `HystrixObservableCommand` 的时候，通过 `HystrixObservableCommand.resumeWithFallback()` 实现服务降级逻辑，该方法会返回一个 `Observable` 对象来发射一个或多个降级结果。

当命令的降级逻辑返回结果之后，Hystrix 就将该结果返回给调用者。当使用 `HystrixCommand.getFallback()` 的时候，它会返回一个 `Observable` 对象，该对象会发射 `getFallback()` 的处理结果。而使用 `HystrixObservableCommand.resumeWithFallback()` 实现的时候，它会将 `Observable` 对象直接返回。

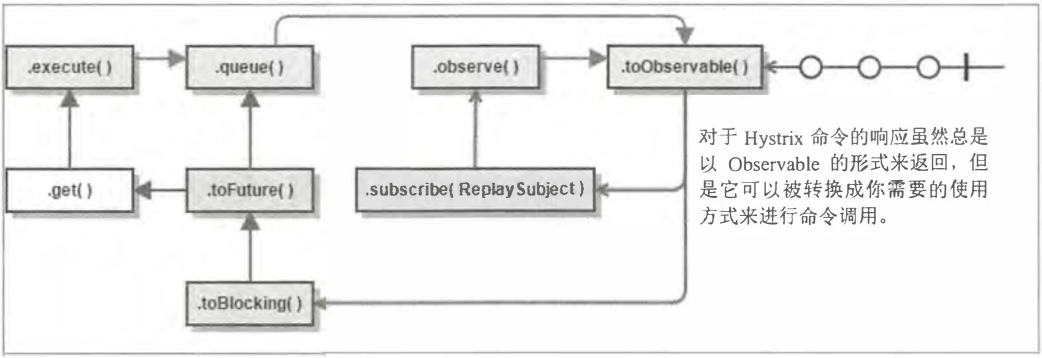
如果我们没有为命令实现降级逻辑或者降级处理逻辑中抛出了异常，Hystrix 依然会返回一个 `Observable` 对象，但是它不会发射任何结果数据，而是通过 `onError` 方法通知命令立即中断请求，并通过 `onError()` 方法将引起命令失败的异常发送给调用者。实现一个有可能失败的降级逻辑是一种非常糟糕的做法，我们应该在实现降级策略时尽可能避免失败的情况。

当然完全不可能出现失败的完美策略是不存在的，如果降级执行发现失败的时候，Hystrix 会根据不同的执行方法做出不同的处理。

- execute()：抛出异常。
- queue()：正常返回 Future 对象，但是当调用 get() 来获取结果的时候会抛出异常。
- observe()：正常返回 Observable 对象，当订阅它的时候，将立即通过调用订阅者的 onError 方法来通知中止请求。
- toObservable()：正常返回 Observable 对象，当订阅它的时候，将通过调用订阅者的 onError 方法来通知中止请求。

9. 返回成功的响应

当 Hystrix 命令执行成功之后，它会将处理结果直接返回或是以 Observable 的形式返回。而具体以哪种方式返回取决于之前第 2 步中我们所提到的对命令的 4 种不同执行方式，下图中总结了这 4 种调用方式之间的依赖关系。我们可以将此图与在第 2 步中对前两者源码的分析联系起来，并且从源头 toObservable() 来开始分析。



- toObservable()：返回最原始的 Observable，必须通过订阅它才会真正触发命令的执行流程。
- observe()：在 toObservable() 产生原始 Observable 之后立即订阅它，让命令能够马上开始异步执行，并返回一个 Observable 对象，当调用它的 subscribe 时，将重新产生结果和通知给订阅者。
- queue()：将 toObservable() 产生的原始 Observable 通过 toBlocking() 方法转换成 BlockingObservable 对象，并调用它的 toFuture() 方法返回异步的 Future 对象。
- execute()：在 queue() 产生异步结果 Future 对象之后，通过调用 get() 方法阻塞并等待结果的返回。

断路器原理

断路器在 HystrixCommand 和 HystrixObservableCommand 执行过程中起到了举足轻重的作用，它是 Hystrix 的核心部件。那么断路器是如何决策熔断和记录信息的呢？

我们先来看看断路器 HystrixCircuitBreaker 的定义：

```
public interface HystrixCircuitBreaker {

  public static class Factory {...}

  static class HystrixCircuitBreakerImpl implements HystrixCircuitBreaker {...}

  static class NoOpCircuitBreaker implements HystrixCircuitBreaker {...}

  public boolean allowRequest();

  public boolean isOpen();

  void markSuccess();
}
```

可以看到它的接口定义并不复杂，主要定义了三个断路器的抽象方法。

- allowRequest()：每个 Hystrix 命令的请求都通过它判断是否被执行。
- isOpen()：返回当前断路器是否打开。
- markSuccess()：用来闭合断路器。

另外还有三个静态类。

- 静态类 Factory 中维护了一个 Hystrix 命令与 HystrixCircuitBreaker 的关系集合：ConcurrentHashMap<String, HystrixCircuitBreaker> circuitBreakersByCommand，其中 String 类型的 key 通过 HystrixCommandKey 定义，每一个 Hystrix 命令需要有一个 key 来标识，同时一个 Hystrix 命令也会在该集合中找到它对应的断路器 HystrixCircuitBreaker 实例。
- 静态类 NoOpCircuitBreaker 定义了一个什么都不做的断路器实现，它允许所有请求，并且断路器状态始终闭合。
- 静态类 HystrixCircuitBreakerImpl 是断路器接口 HystrixCircuitBreaker 的实现类，在该类中定义了断路器的 4 个核心对象。
  - HystrixCommandProperties properties：断路器对应 HystrixCommand 实例的属性对象，它的详细内容我们将在后续章节做具体的介绍。

- HystrixCommandMetrics metrics：用来让 HystrixCommand 记录各类度量指标的对象。
- AtomicBoolean circuitOpen：断路器是否打开的标志，默认为 false。
- AtomicLong circuitOpenedOrLastTestedTime：断路器打开或是上一次测试的时间戳。

HystrixCircuitBreakerImpl 对 HystrixCircuitBreaker 接口的各个方法实现如下所示。

- isOpen()：判断断路器的打开/关闭状态。详细逻辑如下所示。
  - 如果断路器打开标识为 true，则直接返回 true，表示断路器处于打开状态。否则，就从度量指标对象 metrics 中获取 HealthCounts 统计对象做进一步判断(该对象记录了一个滚动时间窗内的请求信息快照，默认时间窗为 10 秒)。
    - 如果它的请求总数（QPS）在预设的阈值范围内就返回 false，表示断路器处于未打开状态。该阈值的配置参数为 circuitBreakerRequestVolumeThreshold，默认值为 20。
    - 如果错误百分比在阈值范围内就返回 false，表示断路器处于未打开状态。该阈值的配置参数为 circuitBreakerErrorThresholdPercentage，默认值为 50。
    - 如果上面的两个条件都不满足，则将断路器设置为打开状态（熔断/短路）。同时，如果是从关闭状态切换到打开状态的话，就将当前时间记录到上面提到的 circuitOpenedOrLastTestedTime 对象中。

```
    public boolean isOpen() {
        if (circuitOpen.get()) {
            return true;
        }
        HealthCounts health = metrics.getHealthCounts();
        if (health.getTotalRequests() <
properties.circuitBreakerRequestVolumeThreshold().get()) {
            return false;
        }
        if (health.getErrorPercentage() <
properties.circuitBreakerErrorThresholdPercentage().get()) {
            return false;
        } else {
```

```
        if (circuitOpen.compareAndSet(false, true)) {
            circuitOpenedOrLastTestedTime.set(System.currentTimeMillis());
            return true;
        } else {
            return true;
        }
    }
}
```

- allowRequest()：判断请求是否被允许，这个实现非常简单。先根据配置对象 properties 中的断路器判断强制打开或关闭属性是否被设置。如果强制打开，就直接返回 false，拒绝请求。如果强制关闭，它会允许所有请求，但是同时也会调用 isOpen()来执行断路器的计算逻辑，用来模拟断路器打开/关闭的行为。在默认情况下，断路器并不会进入这两个强制打开或关闭的分支中去，而是通过!isOpen() || allowSingleTest()来判断是否允许请求访问。!isOpen()之前已经介绍过，用来判断和计算当前断路器是否打开，如果是断开状态就允许请求。那么 allowSingleTest()是用来做什么的呢？

```
@Override
public boolean allowRequest() {
    if (properties.circuitBreakerForceOpen().get()) {
        return false;
    }
    if (properties.circuitBreakerForceClosed().get()) {
        isOpen();
        return true;
    }
    return !isOpen() || allowSingleTest();
}
```

从 allowSingleTest()的实现中我们可以看到，这里使用了在 isOpen()函数中当断路器从闭合到打开时候所记录的时间戳。当断路器在打开状态的时候，这里会判断断开时的时间戳 + 配置中的 circuitBreakerSleepWindowInMilliseconds 时间是否小于当前时间，是的话，就将当前时间更新到记录断路器打开的时间对象 circuitOpenedOrLastTestedTime 中，并且允许此次请求。简单地说，通过 circuitBreakerSleepWindowInMilliseconds 属性设置了一个断路器打开之后的休眠时间（默认为 5 秒），在该休眠时间到达之后，将再次允许请求尝试访问，

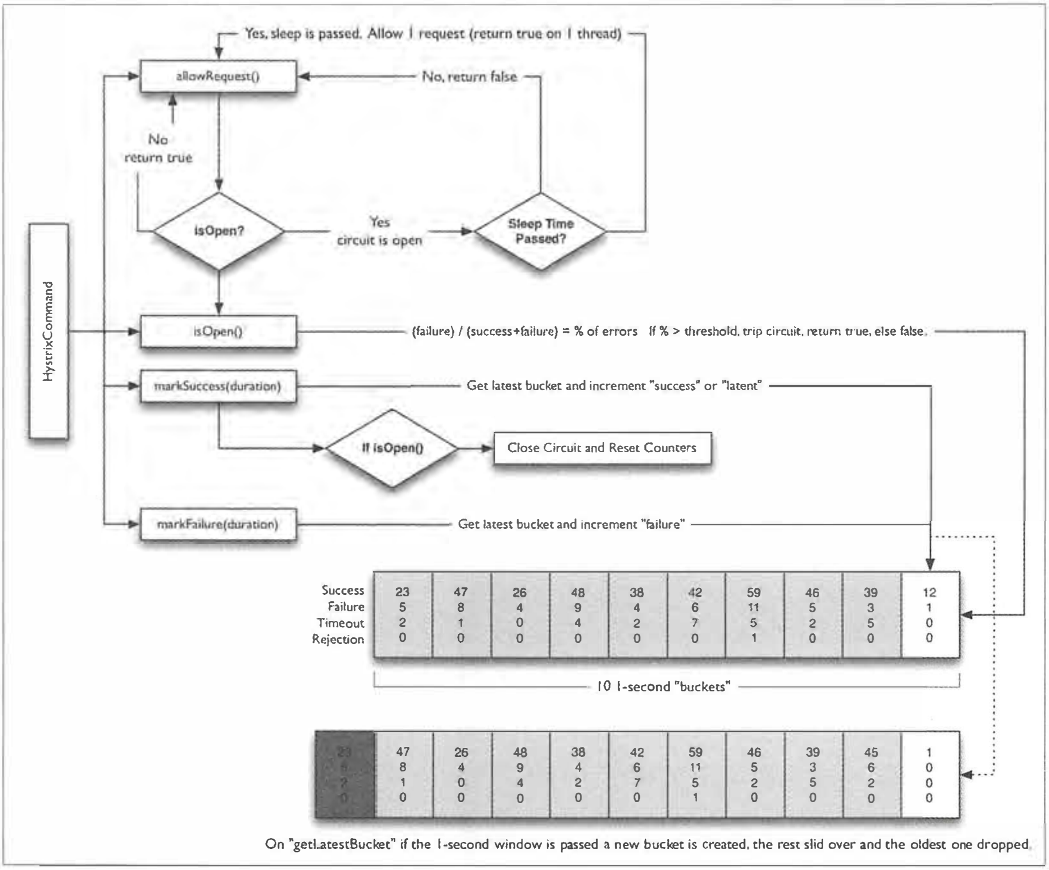
此时断路器处于“半开”状态，若此时请求继续失败，断路器又进入打开状态，并继续等待下一个休眠窗口过去之后再次尝试；若请求成功，则将断路器重新置于关闭状态。所以通过 allowSingleTest()与 isOpen()方法的配合，实现了断路器打开和关闭状态的切换。

```
public boolean allowSingleTest() {
    long timeCircuitOpenedOrWasLastTested = circuitOpenedOrLastTestedTime.get();
    if (circuitOpen.get() && System.currentTimeMillis() >
      timeCircuitOpenedOrWasLastTested +
properties.circuitBreakerSleepWindowInMilliseconds().get()) {
        if (circuitOpenedOrLastTestedTime.compareAndSet(
          timeCircuitOpenedOrWasLastTested, System.currentTimeMillis())) {
            return true;
        }
    }
    return false;
}
```

- markSuccess()：该函数用来在“半开路”状态时使用。若 Hystrix 命令调用成功，通过调用它将打开的断路器关闭，并重置度量指标对象。

```
public void markSuccess() {
    if (circuitOpen.get()) {
        if (circuitOpen.compareAndSet(true, false)) {
            metrics.resetStream();
        }
    }
}
```

下图是 Netflix Hystrix 官方文档中关于断路器的详细执行逻辑，可以帮助我们理解上面的分析内容。



依赖隔离

“舱壁模式”对于熟悉 Docker 的读者一定不陌生，Docker 通过“舱壁模式”实现进程的隔离，使得容器与容器之间不会互相影响。而 Hystrix 则使用该模式实现线程池的隔离，它会为每一个依赖服务创建一个独立的线程池，这样就算某个依赖服务出现延迟过高的情况，也只是对该依赖服务的调用产生影响，而不会拖慢其他的依赖服务。

通过实现对依赖服务的线程池隔离，可以带来如下优势：

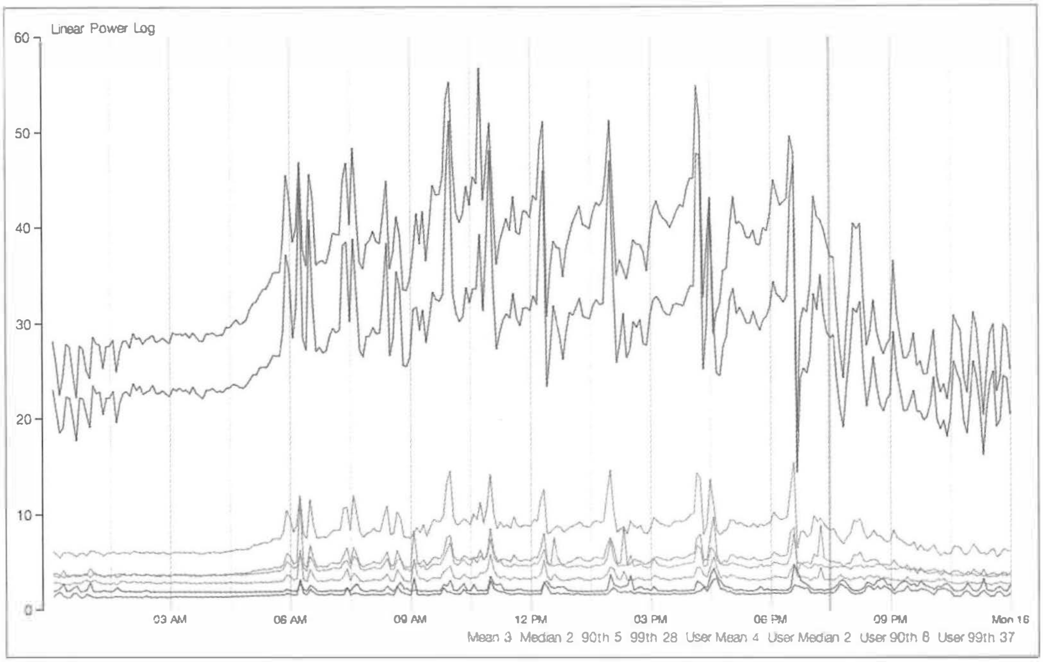
- 应用自身得到完全保护，不会受不可控的依赖服务影响。即便给依赖服务分配的线程池被填满，也不会影响应用自身的其余部分。
- 可以有效降低接入新服务的风险。如果新服务接入后运行不稳定或存在问题，完全不会影响应用其他的请求。

- 当依赖的服务从失效恢复正常后，它的线程池会被清理并且能够马上恢复健康的服务，相比之下，容器级别的清理恢复速度要慢得多。
- 当依赖的服务出现配置错误的时候，线程池会快速反映出此问题（通过失败次数、延迟、超时、拒绝等指标的增加情况）。同时，我们可以在不影响应用功能的情况下通过实时的动态属性刷新（后续会通过 Spring Cloud Config 与 Spring Cloud Bus 的联合使用来介绍）来处理它。
- 当依赖的服务因实现机制调整等原因造成其性能出现很大变化的时候，线程池的监控指标信息会反映出这样的变化。同时，我们也可以通过实时动态刷新自身应用对依赖服务的阈值进行调整以适应依赖方的改变。
- 除了上面通过线程池隔离服务发挥的优点之外，每个专有线程池都提供了内置的并发实现，可以利用它为同步的依赖服务构建异步访问。

总之，通过对依赖服务实现线程池隔离，可让我们的应用更加健壮，不会因为个别依赖服务出现问题而引起非相关服务的异常。同时，也使得我们的应用变得更加灵活，可以在不停止服务的情况下，配合动态配置刷新实现性能配置上的调整。

虽然线程池隔离的方案带来如此多的好处，但是很多使用者可能会担心为每一个依赖服务都分配一个线程池是否会过多地增加系统的负载和开销。对于这一点，使用者不用过于担心，因为这些顾虑也是大部分工程师们会考虑到的，Netflix 在设计 Hystrix 的时候，认为线程池上的开销相对于隔离所带来的好处是无法比拟的。同时，Netflix 也针对线程池的开销做了相关的测试，以用结果打消 Hystrix 实现对性能影响的顾虑。

下图是 Netflix Hystrix 官方提供的一个 Hystrix 命令的性能监控图，该命令以每秒 60 个请求的速度（QPS）对一个单服务实例进行访问，该服务实例每秒运行的线程数峰值为 350 个。



从图中的统计我们可以看到，使用线程池隔离与不使用线程池隔离的耗时差异如下表所示：

比较情况	未使用线程池隔离	使用了线程池隔离	耗时差距
中位数	2ms	2ms	2ms
90 百分位	5ms	8ms	3ms
99 百分位	28ms	37ms	9ms

在 99%的情况下，使用线程池隔离的延迟有 9ms，对于大多数需求来说这样的消耗是微乎其微的，更何况可为系统在稳定性和灵活性上带来巨大的提升。虽然对于大部分的请求我们可以忽略线程池的额外开销，而对于小部分延迟本身就非常小的请求（可能只需要 1ms），那么 9ms 的延迟开销还是非常昂贵的。实际上 Hystrix 也为此设计了另外的解决方案：信号量。

在 Hystrix 中除了可使用线程池之外，还可以使用信号量来控制单个依赖服务的并发度，信号量的开销远比线程池的开销小，但是它不能设置超时和实现异步访问。所以，只有在依赖服务是足够可靠的情况下才使用信号量。在 `HystrixCommand` 和 `HystrixObservableCommand` 中有两处支持信号量的使用。

- **命令执行**：如果将隔离策略参数 `execution.isolation.strategy` 设置为 `SEMAPHORE`，Hystrix 会使用信号量替代线程池来控制依赖服务的并发。

- **降级逻辑**：当 Hystrix 尝试降级逻辑时，它会在调用线程中使用信号量。

信号量的默认值为 10，我们也可以通过动态刷新配置的方式来控制并发线程的数量。对于信号量大小的估算方法与线程池并发度的估算类似。仅访问内存数据的请求一般耗时在 1ms 以内，性能可以达到 5000rps（rps 指每秒的请求数），这样级别的请求可以将信号量设置为 1 或者 2，我们可以按此标准并根据实际请求耗时来设置信号量。

使用详解

在“快速入门”一节中我们已经使用过 Hystrix 中的核心注解`@HystrixCommand`，通过它创建了 `HystrixCommand` 的实现，同时利用 `fallback` 属性指定了服务降级的实现方法。然而这些还只是 Hystrix 使用的一小部分，在实现一个大型分布式系统时，往往还需要更多高级的配置功能。接下来我们将详细介绍 Hystrix 各接口和注解的使用方法。

创建请求命令

Hystrix 命令就是我们之前所说的 `HystrixCommand`，它用来封装具体的依赖服务调用逻辑。

我们可以通过继承的方式来实现，比如：

```
public class UserCommand extends HystrixCommand<User> {

    private RestTemplate restTemplate;
    private Long id;

    public UserCommand(Setter setter, RestTemplate restTemplate, Long id) {
        super(setter);
        this.restTemplate = restTemplate;
        this.id = id;
    }

    @Override
    protected User run() {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

}
```

通过上面实现的 `UserCommand`，我们既可以实现请求的同步执行也可以实现异步执行。

- **同步执行**：`User u = new UserCommand(restTemplate, 1L).execute();`。
- **异步执行**：`Future<User> futureUser = new UserCommand(restTemplate, 1L).queue();`。异步执行的时候，可以通过对返回的 futureUser 调用 get 方法来获取结果。

另外，也可以通过@HystrixCommand 注解来更为优雅地实现 Hystrix 命令的定义，比如：

```
public class UserService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

}
```

虽然@HystrixCommand 注解可以非常优雅地定义 Hystrix 命令的实现，但是如上定义的 getUserById 方式只是同步执行的实现，若要实现异步执行则还需另外定义，比如：

```
@HystrixCommand
public Future<User> getUserByIdAsync(final String id) {
    return new AsyncResult<User>() {
        @Override
        public User invoke() {
            return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
        }
    };
}
```

除了传统的同步执行与异步执行之外，我们还可以将 HystrixCommand 通过 Observable 来实现响应式执行方式。通过调用 observe() 和 toObservable() 方法可以返回 Observable 对象，比如：

```
Observable<String> ho = new UserCommand(restTemplate, 1L).observe();
Observable<String> co = new UserCommand(restTemplate, 1L).toObservable();
```

observe() 和 toObservable() 虽然都返回了 Observable，但是它们略有不同，前者返回的是一个 Hot Observable，该命令会在 observe() 调用的时候立即执行，当 Observable 每次被订阅的时候会重放它的行为；而后者返回的是一个 Cold Observable，

toObservable()执行之后，命令不会被立即执行，只有当所有订阅者都订阅它之后才会执行。更多关于这两个方法的区别可见“原理分析”小节的内容，这里不做具体说明。

虽然 HystrixCommand 具备了 observe()和 toObservable()的功能，但是它的实现有一定的局限性，它返回的 Observable 只能发射一次数据，所以 Hystrix 还提供了另外一个特殊命令封装 HystrixObservableCommand，通过它实现的命令可以获取能发射多次的 Observable。

如果使用 HystrixObservableCommand 来实现命令封装，需要将命令的执行逻辑在 construct 方法中重载，这样 Hystrix 才能将具体逻辑包装到 Observable 内，如下所示：

```
public class UserObservableCommand extends HystrixObservableCommand<User> {

    private RestTemplate restTemplate;
    private Long id;

    public UserObservableCommand(Setter setter, RestTemplate restTemplate, Long id) {
        super(setter);
        this.restTemplate = restTemplate;
        this.id = id;
    }

    @Override
    protected Observable<User> construct() {
        return Observable.create(new Observable.OnSubscribe<User>() {
            @Override
            public void call(Subscriber<? super User> observer) {
                try {
                    if (!observer.isUnsubscribed()) {
                        User user = restTemplate.getForObject("http://USER-SERVICE/
users/{1}", User.class, id);
                        observer.onNext(user);
                        observer.onCompleted();
                    }
                } catch (Exception e) {
                    observer.onError(e);
                }
            }
        } );
    }
}
```

而对此的注解实现依然是使用@HystrixCommand，只是方法定义需要做一些变化，具体内容与 construct()的实现类似，如下所示：

```
    @HystrixCommand
    public Observable<User> getUserById(final String id) {
        return Observable.create(new Observable.OnSubscribe<User>() {
            @Override
            public void call(Subscriber<? super User> observer) {
                try {
                    if (!observer.isUnsubscribed()) {
                        User user = restTemplate.getForObject("http://HELLO-SERVICE/
users/{1}", User.class, id);
                        observer.onNext(user);
                        observer.onCompleted();
                    }
                } catch (Exception e) {
                    observer.onError(e);
                }
            }
        });
    }
```

在使用@HystrixCommand注解实现响应式命令时，可以通过observable-ExecutionMode参数来控制是使用observe()还是toObservable()的执行方式。该参数有下面两种设置方式。

- @HystrixCommand(observableExecutionMode = ObservableExecution-Mode.EAGER)：EAGER是该参数的模式值，表示使用observe()执行方式。
- @HystrixCommand(observableExecutionMode = ObservableExecution-Mode.LAZY)：表示使用toObservable()执行方式。

定义服务降级

fallback是Hystrix命令执行失败时使用的后备方法，用来实现服务的降级处理逻辑。在HystrixCommand中可以通过重载getFallback()方法来实现服务降级逻辑，Hystrix会在run()执行过程中出现错误、超时、线程池拒绝、断路器熔断等情况时，执行getFallback()方法内的逻辑，比如我们可以用如下方式实现服务降级逻辑：

```
    public class UserCommand extends HystrixCommand<User> {

        private RestTemplate restTemplate;
        private Long id;

        public UserCommand(Setter setter, RestTemplate restTemplate, Long id) {
            super(setter);
            this.restTemplate = restTemplate;
            this.id = id;
```

```
    }

    @Override
    protected User run() {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}", User.class, id);
    }

    @Override
    protected User getFallback() {
        return new User();
    }

}
```

在 HystrixObservableCommand 实现的 Hystrix 命令中，我们可以通过重载 resumeWithFallback 方法来实现服务降级逻辑。该方法会返回一个 Observable 对象，当命令执行失败的时候，Hystrix 会将 Observable 中的结果通知给所有的订阅者。

若要通过注解实现服务降级只需要使用@HystrixCommand 中的 fallbackMethod 参数来指定具体的服务降级实现方法，如下所示：

```
public class UserService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "defaultUser")
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

    public User defaultUser() {
        return new User();
    }
}
```

在使用注解来定义服务降级逻辑时，我们需要将具体的 Hystrix 命令与 fallback 实现函数定义在同一个类中，并且 fallbackMethod 的值必须与实现 fallback 方法的名字相同。由于必须定义在一个类中，所以对于 fallback 的访问修饰符没有特定的要求，定义为 private、protected、public 均可。

在上面的例子中，defaultUser 方法将在 getUserById 执行时发生错误的情况下被执行。若 defaultUser 方法实现的并不是一个稳定逻辑，它依然可能会发生异常，那么我们也可以为它添加@HystrixCommand 注解以生成 Hystrix 命令，同时使用 fallbackMethod

来指定服务降级逻辑，比如：

```
public class UserService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "defaultUser")
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

    @HystrixCommand(fallbackMethod = "defaultUserSec")
    public User defaultUser() {
        // 此处可能是另外一个网络请求来获取，所以也有可能失败
        return new User("First Fallback");
    }

    public User defaultUserSec() {
        return new User("Second Fallback");
    }
}
```

在实际使用时，我们需要为大多数执行过程中可能会失败的 Hystrix 命令实现服务降级逻辑，但是也有一些情况可以不去实现降级逻辑，如下所示。

- **执行写操作的命令**：当 Hystrix 命令是用来执行写操作而不是返回一些信息的时候，通常情况下这类操作的返回类型是 `void` 或是为空的 `Observable`，实现服务降级的意义不是很大。当写入操作失败的时候，我们通常只需要通知调用者即可。
- **执行批处理或离线计算的命令**：当 Hystrix 命令是用来执行批处理程序生成一份报告或是进行任何类型的离线计算时，那么通常这些操作只需要将错误传播给调用者，然后让调用者稍后重试而不是发送给调用者一个静默的降级处理响应。

不论 Hystrix 命令是否实现了服务降级，命令状态和断路器状态都会更新，并且我们可以由此了解到命令执行的失败情况。

异常处理

异常传播

在 HystrixCommand 实现的 run()方法中抛出异常时，除了 HystrixBadRequest-Exception 之外，其他异常均会被 Hystrix 认为命令执行失败并触发服务降级的处理逻辑，所以当需要在命令执行中抛出不触发服务降级的异常时来使用它。

而在使用注册配置实现 Hystrix 命令时，它还支持忽略指定异常类型功能，只需要通过设置@HystrixCommand 注解的 ignoreExceptions 参数，比如：

```
@HystrixCommand(ignoreExceptions = {BadRequestException.class})
public User getUserById(Long id) {
    return restTemplate.getForObject("http://USER-SERVICE/users/{1}", User.class,
id);
}
```

如上面代码的定义，当 getUserById 方法抛出了类型为 BadRequestException 的异常时，Hystrix 会将它包装在 HystrixBadRequestException 中抛出，这样就不会触发后续的 fallback 逻辑。

异常获取

当 Hystrix 命令因为异常（除了 HystrixBadRequestException 的异常）进入服务降级逻辑之后，往往需要对不同异常做针对性的处理，那么我们如何来获取当前抛出的异常呢？

在以传统继承方式实现的 Hystrix 命令中，我们可以用 getFallback()方法通过 Throwable getExecutionException()方法来获取具体的异常，通过判断来进入不同的处理逻辑。

除了传统的实现方式之外，注解配置方式也同样可以实现异常的获取。它的实现也非常简单，只需要在 fallback 实现方法的参数中增加 Throwable e 对象的定义，这样在方法内部就可以获取触发服务降级的具体异常内容了，比如：

```
@HystrixCommand(fallbackMethod = "fallback1")
User getUserById(String id) {
    throw new RuntimeException("getUserById command failed");
}

User fallback1(String id, Throwable e) {
    assert "getUserById command failed".equals(e.getMessage());
}
```

命令名称、分组以及线程池划分

以继承方式实现的 Hystrix 命令使用类名作为默认的命令名称，我们也可以在构造函数中通过 Setter 静态类来设置，比如：

```
    public UserCommand() {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("GroupName"))
          .andCommandKey(HystrixCommandKey.Factory.asKey("CommandName")););
    }
```

从上面 Setter 的使用中可以看到，我们并没有直接设置命令名称，而是先调用了 withGroupKey 来设置命令组名，然后才通过调用 andCommandKey 来设置命令名。这是因为在 Setter 的定义中，只有 withGroupKey 静态函数可以创建 Setter 的实例，所以 GroupKey 是每个 Setter 必需的参数，而 CommandKey 则是一个可选参数。

通过设置命令组，Hystrix 会根据组来组织和统计命令的告警、仪表盘等信息。那么为什么一定要设置命令组呢？因为除了根据组能实现统计之外，Hystrix 命令默认的线程划分也是根据命令分组来实现的。默认情况下，Hystrix 会让相同组名的命令使用同一个线程池，所以我们需要在创建 Hystrix 命令时为其指定命令组名来实现默认的线程池划分。

如果 Hystrix 的线程池分配仅仅依靠命令组来划分，那么它就显得不够灵活了，所以 Hystrix 还提供了 HystrixThreadPoolKey 来对线程池进行设置，通过它我们可以实现更细粒度的线程池划分，比如：

```
    public UserCommand() {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey
("CommandGroupKey"))
              .andCommandKey(HystrixCommandKey.Factory.asKey("CommandKey"))
              .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey("ThreadPoolKey")));
    }
```

如果在没有特别指定 HystrixThreadPoolKey 的情况下，依然会使用命令组的方式来划分线程池。通常情况下，尽量通过 HystrixThreadPoolKey 的方式来指定线程池的划分，而不是通过组名的默认方式实现划分，因为多个不同的命令可能从业务逻辑上来看属于同一个组，但是往往从实现本身上需要跟其他命令进行隔离。

上面已经介绍了如何为通过继承实现的 HystrixCommand 设置命令名称、分组以及线程池划分，那么当我们使用@HystrixCommand 注解的时候，又该如何设置呢？只需设置@HystrixCommand 注解的 commandKey、groupKey 以及 threadPoolKey 属性即可，它们分别表示了命令名称、分组以及线程池划分，比如我们可以像下面这样进行设置：

```
    @HystrixCommand(commandKey = "getUserById", groupKey = "UserGroup", threadPoolKey
= "getUserByIdThread")
    public User getUserById(Long id) {
```

```
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}", User.class,
id);
    }
```

请求缓存

当系统用户不断增长时，每个微服务需要承受的并发压力也越来越大。在分布式环境下，通常压力来自于对依赖服务的调用，因为请求依赖服务的资源需要通过通信来实现，这样的依赖方式比起进程内的调用方式会引起一部分的性能损失，同时 HTTP 相比于其他高性能的通信协议在速度上没有任何优势，所以它有些类似于对数据库这样的外部资源进行读写操作，在高并发的情况下可能会成为系统的瓶颈。既然如此，我们很容易地可以联想到，类似数据访问的缓存保护是否也可以应用到依赖服务的调用上呢？

答案显而易见，在高并发的场景之下，Hystrix 中提供了请求缓存的功能，我们可以方便地开启和使用请求缓存来优化系统，达到减轻高并发时的请求线程消耗、降低请求响应时间的效果。

开启请求缓存功能

Hystrix 请求缓存的使用非常简单，我们只需要在实现 `HystrixCommand` 或 `HystrixObservableCommand` 时，通过重载 `getCacheKey()`方法来开启请求缓存，比如：

```
public class UserCommand extends HystrixCommand<User> {

    private RestTemplate restTemplate;
    private Long id;

    public UserCommand(RestTemplate restTemplate, Long id) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey
("UserGroup")));
        this.restTemplate = restTemplate;
        this.id = id;
    }

    @Override
    protected User run() {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

    @Override
    protected String getCacheKey() {
        return String.valueOf(id);
```

```
        }

    }
```

在上面的例子中，我们通过在 `getCacheKey` 方法中返回的请求缓存 key 值（使用了传入的获取 User 对象的 id 值），就能让该请求命令具备缓存功能。此时，当不同的外部请求处理逻辑调用了同一个依赖服务时，Hystrix 会根据 `getCacheKey` 方法返回的值来区分是否是重复的请求，如果它们的 cacheKey 相同，那么该依赖服务只会在第一个请求到达时被真实地调用一次，另外一个请求则是直接从请求缓存中返回结果，所以通过开启请求缓存可以让我们实现的 Hystrix 命令具备下面几项好处：

- 减少重复的请求数，降低依赖服务的并发度。
- 在同一用户请求的上下文中，相同依赖服务的返回数据始终保持一致。
- 请求缓存在 `run()` 和 `construct()` 执行之前生效，所以可以有效减少不必要的线程开销。

清理失效缓存功能

使用请求缓存时，如果只是读操作，那么不需要考虑缓存内容是否正确的问题，但是如果请求命令中还有更新数据的写操作，那么缓存中的数据就需要我们在进行写操作时进行及时处理，以防止读操作的请求命令获取到了失效的数据。

在 Hystrix 中，我们可以通过 `HystrixRequestCache.clear()` 方法来进行缓存的清理，具体示例如下：

```
    public class UserGetCommand extends HystrixCommand<User> {

        private static final HystrixCommandKey GETTER_KEY = HystrixCommandKey.Factory.
asKey("CommandKey");
        private RestTemplate restTemplate;
        private Long id;

        public UserGetCommand(RestTemplate restTemplate, Long id) {
            super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey
("GetSetGet"))
                    .andCommandKey(GETTER_KEY));
            this.restTemplate = restTemplate;
            this.id = id;
        }

        @Override
        protected User run() {
            return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
```

```
        }

        @Override
        protected String getCacheKey() {
            // 根据id置入缓存
            return String.valueOf(id);
        }

        public static void flushCache(Long id) {
            // 刷新缓存，根据id进行清理
            HystrixRequestCache.getInstance(GETTER_KEY,
                    HystrixConcurrencyStrategyDefault.getInstance()).clear(String.
valueOf(id));
        }

    }

    public class UserPostCommand extends HystrixCommand<User> {

        private RestTemplate restTemplate;
        private User user;

        public UserPostCommand(RestTemplate restTemplate, User user) {

super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey("GetSetGet")));
            this.restTemplate = restTemplate;
            this.user = user;
        }

        @Override
        protected User run() {
            // 写操作
            User r = restTemplate.postForObject("http://USER-SERVICE/users", user,
User.class);
            // 刷新缓存，清理缓存中失效的User
            UserGetCommand.flushCache(user.getId());
            return r;
        }

    }
```

该示例中主要有两个请求命令：`UserGetCommand` 用于根据 id 获取 User 对象、而 `UserPostCommand` 用于更新 User 对象。当我们对 `UserGetCommand` 命令实现了请求缓存之后，那么势必需要为 `UserPostCommand` 命令实现缓存的清理，以保证 User 被更新

之后，Hystrix 请求缓存中相同缓存 Key 的结果被移除，这样在下一次获取 User 的时候不会从缓存中获取到未更新的结果。

我们可以看到，在上面 UserGetCommand 的实现中，增加了一个静态方法 flushCache，该方法通过 HystrixRequestCache.getInstance(GETTER_KEY, HystrixConcurrencyStrategyDefault.getInstance()) 方法从默认的 Hystrix 并发策略中根据 GETTER_KEY 获取到该命令的请求缓存对象 HystrixRequestCache 的实例，然后再调用该请求缓存对象实例的 clear 方法，对 Key 为更新 User 的 id 值的缓存内容进行清理。而在 UserPostCommand 的实现中，在 run 方法调用依赖服务之后，增加了对 UserGetCommand 中静态方法 flushCache 的调用，以实现对失效缓存的清理。

工作原理

通过上面的入门例子，我们已经能够体会到在 Hystrix 中实现请求缓存是非常方便的，那么它是如何做到的呢？我们不妨通过分析其源码来了解一下它的实现原理，对其有一个深入的理解，有助于指导我们正确使用和配置请求缓存。由于 getCacheKey 方法在 AbstractCommand 抽象命令类中实现，所以我们可以先从这个抽象命令类的实现中看起。

从下面 AbstractCommand 的源码片段中，我们可以看到，getCacheKey 方法默认返回的是 null，并且从 isRequestCachingEnabled 方法的实现逻辑中我们还可以知道，如果不重写 getCacheKey 方法，让它返回一个非 null 值，那么缓存功能是不会开启的；同时请求命令的缓存开启属性也需要设置为 true 才能开启（该属性默认为 true，所以通常用该属性来控制请求缓存功能的强制关闭）。

```
    abstract class AbstractCommand<R> implements HystrixInvokableInfo<R>,
HystrixObservable<R> {
        ...
        protected final HystrixRequestCache requestCache;
        ...
        protected String getCacheKey() {
            return null;
        }
        protected boolean isRequestCachingEnabled() {
            return properties.requestCacheEnabled().get() && getCacheKey() != null;
        }
        ...
        public Observable<R> toObservable() {
            ...
            // 尝试从缓存中获取结果
```

```
        final boolean requestCacheEnabled = isRequestCachingEnabled();
        final String cacheKey = getCacheKey();
        final AbstractCommand<R> _cmd = this;
        if (requestCacheEnabled) {
            HystrixCommandResponseFromCache<R> fromCache =
(HystrixCommandResponseFromCache<R>) requestCache.get(cacheKey);
            if (fromCache != null) {
                isResponseFromCache = true;
                return handleRequestCacheHitAndEmitValues(fromCache, _cmd);
            }
        }

        ...

        Observable<R> hystrixObservable =
Observable.defer(applyHystrixSemantics).map(wrapWithAllOnNextHooks);
        Observable<R> afterCache;

        // 加入缓存
        if (requestCacheEnabled && cacheKey != null) {
            HystrixCachedObservable<R> toCache =
HystrixCachedObservable.from(hystrixObservable, this);
            HystrixCommandResponseFromCache<R> fromCache =
(HystrixCommandResponseFromCache<R>) requestCache.putIfAbsent(cacheKey, toCache);
            if (fromCache != null) {
                toCache.unsubscribe();
                isResponseFromCache = true;
                return handleRequestCacheHitAndEmitValues(fromCache, _cmd);
            } else {
                afterCache = toCache.toObservable();
            }
        } else {
            afterCache = hystrixObservable;
        }
    }
    ...
}
```

另外，从命令异步执行的核心方法 toObservable() 中，我们可以看到与缓存相关的主要执行步骤，它分为两部分内容：尝试获取请求缓存以及将请求结果加入缓存。

- **尝试获取请求缓存：** Hystrix 命令在执行前会根据之前提到的 isRequestCachingEnabled 方法来判断当前命令是否启用了请求缓存。如果开启了请求缓存并且重写了 getCacheKey 方法，并返回了一个非 null 的缓存 Key 值，那么就使用 getCacheKey 返回的 Key 值去调用 HystrixRequestCache 中的 get(String cacheKey)来获取缓存的 HystrixCachedObservable 对象。
- **将请求结果加入缓存：** 在执行命令缓存操作之前，我们可以看到已经获得了一个延迟执行的命令结果对象 hystrixObservable。接下来与尝试获取请求缓存操作一样，需要先判断当前命令是否开启了请求缓存功能，如果开启了请求缓存并且 getCacheKey 返回了具体的 Key 值，就将 hystrixObservable 对象包装成请求缓存结果 HystrixCachedObservable 的实例对象 toCache，然后将其放入当前命令的缓存对象中。从调用的方法 putIfAbsent 中，我们大致可以猜到在请求缓存对象 HystrixRequestCache 中维护了一个线程安全的 Map 来保存请求缓存的响应，所以在调用 putIfAbsent 将包装的请求缓存放入缓存对象后，对其返回结果 fromCache 进行了判断，如果其不为 null，说明当前缓存 Key 的请求命令缓存命中，直接对 toCache 执行取消订阅操作（即，不再发起真实请求），同时调用缓存命令的处理方法 handleRequestCacheHitAndEmitValues 来执行缓存命中的结果获取。如果返回的 fromCache 为 null 说明缓存没有命中，则将当前结果 toCache 缓存起来，并将其转换成 Observable 返回给调用者使用。

使用注解实现请求缓存

Hystrix 的请求缓存除了可以通过上面传统的方式实现之外，还可以通过注解的方式进行配置实现。注解配置的定义实现同 JSR 107 的定义非常相似，但由于 Hystrix 不需要独立外置的缓存系统来支持，所以没有 JSR 107 的定义那么复杂，它只提供了三个专用于请求缓存的注解。

注解	描述	属性
@CacheResult	该注解用来标记请求命令返回的结果应该被缓存，它必须与@HystrixCommand 注解结合使用	cacheKeyMethod
@CacheRemove	该注解用来让请求命令的缓存失效，失效的缓存根据定义的 Key 决定	commandKey, cacheKeyMethod
@CacheKey	该注解用来在请求命令的参数上标记，使其作为缓存的 Key 值，如果没有标注则会使用所有参数。如果同时还使用了@CacheResult 和@CacheRemove 注解的 cacheKeyMethod 方法指定缓存 Key 的生成，那么该注解将不会起作用	value

JSR 107 是 Java 缓存 API 的定义，也被称为 JCache。它定义了一系列开发人员使用的标准化 Java 缓存 API 和服务提供商使用的标准 SPI。

下面我们从几个方面的实例来看看这几个注解的具体使用方法。

- **设置请求缓存**：通过注解为请求命令开启缓存功能非常简单，如下例所示，我们只需添加@CacheResult 注解即可。当该依赖服务被调用并返回 User 对象时，由于该方法被@CacheResult 注解修改，所以 Hystrix 会将该结果置入请求缓存中，而它的缓存 Key 值会使用所有的参数，也就是这里 Long 类型的 id 值。

```
    @CacheResult
    @HystrixCommand
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }
```

- **定义缓存 Key**：当使用注解来定义请求缓存时，若要为请求命令指定具体的缓存 Key 生成规则，我们可以使用@CacheResult 和@CacheRemove 注解的 cacheKeyMethod 方法来指定具体的生成函数；也可以通过使用@CacheKey 注解在方法参数中指定用于组装缓存 Key 的元素。

使用 cacheKeyMethod 方法的示例如下，它通过在请求命令的同一个类中定义一个专门生成 Key 的方法，并用@CacheResult 注解的 cacheKeyMethod 方法来指定它即可。它的配置方式类似于@HystrixCommand 服务降级 fallbackMethod 的使用。

```
    @CacheResult(cacheKeyMethod = "getUserByIdCacheKey")
    @HystrixCommand
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

    private Long getUserByIdCacheKey(Long id) {
        return id;
    }
```

通过@CacheKey 注解实现的方式更加简单，具体示例如下所示。但是在使用@CacheKey 注解的时候需要注意，它的优先级比 cacheKeyMethod 的优先级低，如果已经使用了 cacheKeyMethod 指定缓存 Key 的生成函数，那么@CacheKey 注解不会生效。

```
    @CacheResult
    @HystrixCommand
    public User getUserById(@CacheKey("id") Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
```

```
User.class, id);
    }
```

@CacheKey 注解除了可以指定方法参数作为缓存 Key 之外，它还允许访问参数对象的内部属性作为缓存 Key。比如下面的例子，它指定了 User 对象的 id 属性作为缓存 Key。

```
    @CacheResult
    @HystrixCommand
    public User getUserById(@CacheKey("id") User user) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}", User.class,
user.getId());
    }
```

- **缓存清理**：在之前的例子中，我们已经通过@CacheResult 注解将请求结果置入 Hystrix 的请求缓存之中。若该内容调用了 update 操作进行了更新，那么此时请求缓存中的结果与实际结果就会产生不一致（缓存中的结果实际上已经失效了），所以我们需要在 update 类型的操作上对失效的缓存进行清理。在 Hystrix 的注解配置中，可以通过@CacheRemove 注解来实现失效缓存的清理，比如下面的例子所示：

```
    @CacheResult
    @HystrixCommand
    public User getUserById(@CacheKey("id") Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}", User.class,
id);
    }

    @CacheRemove(commandKey = "getUserById")
    @HystrixCommand
    public void update(@CacheKey("id") User user) {
        return restTemplate.postForObject("http://USER-SERVICE/users", user,
User.class);
    }
```

需要注意的是，@CacheRemove 注解的 commandKey 属性是必须要指定的，它用来指明需要使用请求缓存的请求命令，因为只有通过该属性的配置，Hystrix 才能找到正确的请求命令缓存位置。

请求合并

微服务架构中的依赖通常通过远程调用实现，而远程调用中最常见的问题就是通信消耗与连接数占用。在高并发的情况之下，因通信次数的增加，总的通信时间消耗将会变得不那么理想。同时，因为依赖服务的线程池资源有限，将出现排队等待与响应延迟的情况。为了优化这两个问题，Hystrix 提供了 HystrixCollapser 来实现请求的合并，以减少通信消耗和线程数的占用。

HystrixCollapser 实现了在 HystrixCommand 之前放置一个合并处理器，将处于一个很短的时间窗（默认 10 毫秒）内对同一依赖服务的多个请求进行整合并以批量方式发起请求的功能（服务提供方也需要提供相应的批量实现接口）。通过 HystrixCollapser 的封装，开发者不需要关注线程合并的细节过程，只需关注批量化服务和处理。下面我们从 HystrixCollapser 的使用实例中对其合并请求的过程一探究竟。

```
    public abstract class HystrixCollapser<BatchReturnType, ResponseType,
RequestArgumentType> implements
            HystrixExecutable<ResponseType>, HystrixObservable<ResponseType> {
        ...
        public abstract RequestArgumentType getRequestArgument();

        protected abstract HystrixCommand<BatchReturnType>
createCommand(Collection<CollapsedRequest<ResponseType, RequestArgumentType>>
requests);

        protected abstract void mapResponseToRequests(BatchReturnType batchResponse,
Collection<CollapsedRequest<ResponseType, RequestArgumentType>> requests);
        ...
    }
```

从 HystrixCollapser 抽象类的定义中可以看到，它指定了三个不同的类型。

- BatchReturnType：合并后批量请求的返回类型。
- ResponseType：单个请求返回的类型。
- RequestArgumentType：请求参数类型。

而对于这三个类型的使用可以在它的三个抽象方法中看到。

- RequestArgumentType getRequestArgument()：该函数用来定义获取请求参数的方法。
- HystrixCommand<BatchReturnType>createCommand(Collection<CollapsedRequest<ResponseType, RequestArgumentType>> requests)：合并请求产生批量命令的具体实现方法。
- mapResponseToRequests(BatchReturnType batchResponse, Collection<CollapsedRequest<ResponseType, RequestArgumentType>> requests)：批量命令结果返回后的处理，这里需要实现将批量结果拆分并传递给合并前的各个原子请求命令的逻辑。

接下来，我们通过一个简单的示例来直观理解实现请求合并的过程。

假设当前微服务 USER-SERVICE 提供了两个获取 User 的接口。

- `/users/{id}`：根据 id 返回 User 对象的 GET 请求接口。
- `/users?ids={ids}`：根据 ids 返回 User 对象列表的 GET 请求接口，其中 ids 为以逗号分隔的 id 集合。

而在服务消费端，已经为这两个远程接口通过 `RestTemplate` 实现了简单的调用，具体如下所示：

```
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private RestTemplate restTemplate;

    @Override
    public User find(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}",
User.class, id);
    }

    @Override
    public List<User> findAll(List<Long> ids) {
        return restTemplate.getForObject("http://USER-SERVICE/users?ids={1}",
List.class, StringUtils.join(ids, ","));
    }

}
```

接着，我们实现将短时间内多个获取单一 User 对象的请求命令进行合并。

- 第一步，为请求合并的实现准备一个批量请求命令的实现，具体如下：

```
public class UserBatchCommand extends HystrixCommand<List<User>> {

    UserService userService;
    List<Long> userIds;

    public UserBatchCommand(UserService userService, List<Long> userIds) {
        super(Setter.withGroupKey(asKey("userServiceCommand")));
        this.userIds = userIds;
        this.userService = userService;
    }

    @Override
    protected List<User> run() throws Exception {
```

```
        return userService.findAll(userIds);
    }

}
```

批量请求命令实际上就是一个简单的 HystrixCommand 实现，从上面的实现中可以看到它通过调用 `userService.findAll` 方法来访问`/users?ids={ids}`接口以返回 User 的列表结果。

- 第二步，通过继承 `HystrixCollapser` 实现请求合并器：

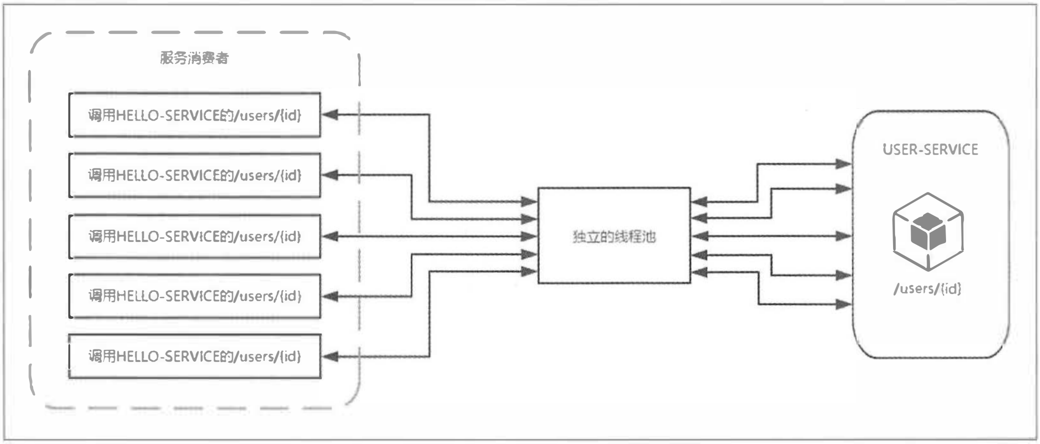
```
public class UserCollapseCommand extends HystrixCollapser<List<User>, User, Long> {

    private UserService userService;
    private Long userId;

    public UserCollapseCommand(UserService userService, Long userId) {
        super(Setter.withCollapserKey(HystrixCollapserKey.Factory.asKey
("userCollapseCommand")).andCollapserPropertiesDefaults(

HystrixCollapserProperties.Setter().withTimerDelayInMilliseconds(100)));
        this.userService = userService;
        this.userId = userId;
    }

    @Override
    public Long getRequestArgument() {
        return userId;
    }

    @Override
    protected HystrixCommand<List<User>>
createCommand(Collection<CollapsedRequest<User, Long>> collapsedRequests) {
        List<Long> userIds = new ArrayList<>(collapsedRequests.size());

userIds.addAll(collapsedRequests.stream().map(CollapsedRequest::getArgument).collec
t(Collectors.toList()));
        return new UserBatchCommand(userService, userIds);
    }

    @Override
    protected void mapResponseToRequests(List<User> batchResponse,
Collection<CollapsedRequest<User, Long>> collapsedRequests) {
        int count = 0;
        for (CollapsedRequest<User, Long> collapsedRequest : collapsedRequests) {
            User user = batchResponse.get(count++);
```

```
            collapsedRequest.setResponse(user);
        }
    }

}
```

在上面的构造函数中，我们为请求合并器设置了时间延迟属性，合并器会在该时间窗内收集获取单个 User 的请求并在时间窗结束时进行合并组装成单个批量请求。getRequestArgument方法返回给定的单个请求参数 userId，而 createCommand 和 mapResponseToRequests 是请求合并器的两个核心。

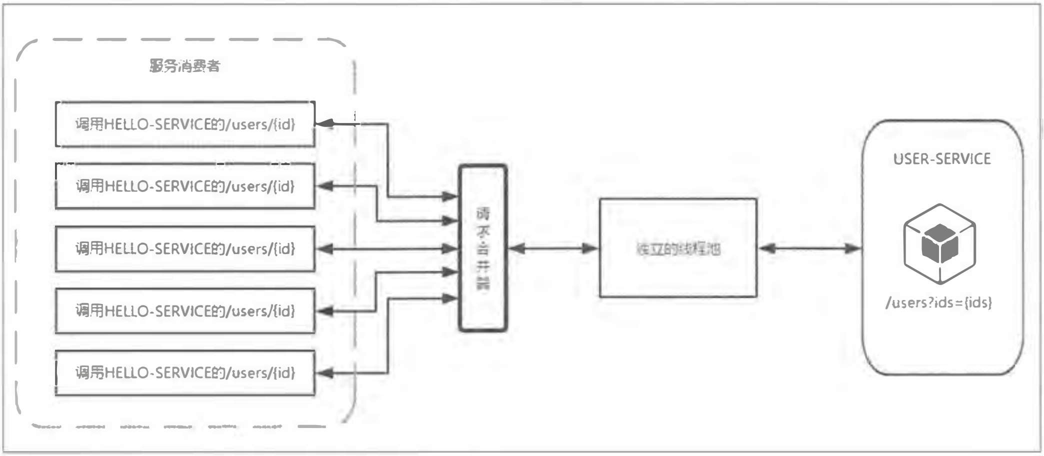
- createCommand：该方法的 collapsedRequests 参数中保存了延迟时间窗中收集到的所有获取单个 User 的请求。通过获取这些请求的参数来组织上面我们准备的批量请求命令 UserBatchCommand 实例。
- mapResponseToRequests：在批量请求命令 UserBatchCommand 实例被触发执行完成之后，该方法开始执行，其中 batchResponse 参数保存了 createCommand 中组织的批量请求命令的返回结果，而 collapsedRequests 参数则代表了每个被合并的请求。在这里我们通过遍历批量结果 batchResponse 对象，为 collapsedRequests 中每个合并前的单个请求设置返回结果，以此完成批量结果到单个请求结果的转换。

下图展示了在未使用 HystrixCollapser 请求合并器之前的线程使用情况。可以看到，当服务消费者同时对 USER-SERVICE 的/users/{id}接口发起了 5 个请求时，会向该依赖服务的独立线程池中申请 5 个线程来完成各自的请求操作。



而在使用了 HystrixCollapser 请求合并器之后，相同情况下的线程占用如下图所示。由于同一时间发生的 5 个请求处于请求合并器的一个时间窗内，这些发向/users/{id}

接口的请求被请求合并器拦截下来，并在合并器中进行组合，然后将这些请求合并成一个请求发向 USER-SERVICE 的批量接口/users?ids={ids}。在获取到批量请求结果之后，通过请求合并器再将批量结果拆分并分配给每个被合并的请求。从图中我们可以看到，通过使用请求合并器有效减少了对线程池中资源的占用。所以在资源有效并且短时间内会产生高并发请求的时候，为避免连接不够用而引起的延迟可以考虑使用请求合并器的方式来处理和优化。



使用注解实现请求合并器

在快速入门的例子中，我们使用@HystrixCommand 注解优雅地实现了 HystrixCommand 的定义，那么对于请求合并器是否也可以通过注解来定义呢？答案是肯定的！

以上面实现的请求合并器为例，还可以通过如下方式实现：

```
@Service
public class UserService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCollapser(batchMethod = "findAll", collapserProperties = {
            @HystrixProperty(name="timerDelayInMilliseconds", value = "100")
    })
    public User find(Long id) {
        return null;
    }

    @HystrixCommand
```

```
    public List<User> findAll(List<Long> ids) {
        return restTemplate.getForObject("http://USER-SERVICE/users?ids={1}",
List.class, StringUtils.join(ids, ","));
    }
}
```

我们之前已经介绍过@HystrixCommand 了，可以看到，这里通过它定义了两个 Hystrix 命令，一个用于请求/users/{id}接口，一个用于请求/users?ids={ids}接口。而在请求/users/{id}接口的方法上通过@HystrixCollapser 注解为其创建了合并请求器，通过 batchMethod 属性指定了批量请求的实现方法为 findAll 方法（即请求/users?ids={ids}接口的命令），同时通过 collapserProperties 属性为合并请求器设置了相关属性，这里使用@HystrixProperty(name="timerDelayInMilliseconds", value = "100")将合并时间窗设置为 100 毫秒。这样通过@HystrixCollapser 注解简单而又优雅地实现了在/users/{id}依赖服务之前设置了一个批量请求合并器。

请求合并的额外开销

虽然通过请求合并可以减少请求的数量以缓解依赖服务线程池的资源，但是在使用的时候也需要注意它所带来的额外开销：用于请求合并的延迟时间窗会使得依赖服务的请求延迟增高。比如，某个请求不通过请求合并器访问的平均耗时为 5ms，请求合并的延迟时间窗为 10ms（默认值），那么当该请求设置了请求合并器之后，最坏情况下（在延迟时间窗结束时才发起请求）该请求需要 15ms 才能完成。

由于请求合并器的延迟时间窗会带来额外开销，所以我们是否使用请求合并器需要根据依赖服务调用的实际情况来选择，主要考虑下面两个方面。

- **请求命令本身的延迟。**如果依赖服务的请求命令本身是一个高延迟的命令，那么可以使用请求合并器，因为延迟时间窗的时间消耗显得微不足道了。
- **延迟时间窗内的并发量。**如果一个时间窗内只有 1~2 个请求，那么这样的依赖服务不适合使用请求合并器。这种情况不但不能提升系统性能，反而会成为系统瓶颈，因为每个请求都需要多消耗一个时间窗才响应。相反，如果一个时间窗内具有很高的并发量，并且服务提供方也实现了批量处理接口，那么使用请求合并器可以有效减少网络连接数量并极大提升系统吞吐量，此时延迟时间窗所增加的消耗就可以忽略不计了。

属性详解

在之前介绍 Hystrix 的使用方法时，已经涉及过一些 Hystrix 属性的配置，我们可以根

据实现 HystrixCommand 的不同方式将配置方法分为如下两类。

- 当通过继承的方式实现时，可使用 Setter 对象来对请求命令的属性进行设置，比如下面的例子：

```
    public HystrixCommandInstance(int id) {
        super(Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey
("ExampleGroup"))
            .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()
                .withExecutionTimeoutInMilliseconds(500)));
        this.id = id;
    }
```

- 当通过注解的方法实现时，只需使用@HystrixCommand 中的 command-Properties 属性来设置，比如：

```
    @HystrixCommand(commandKey = "helloKey",
        commandProperties = {
            @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",
value = "5000")
        }
    )
    public User getUserById(Long id) {
        return restTemplate.getForObject("http://USER-SERVICE/users/{1}", User.class,
id);
    }
```

实际上，Hystrix 为我们提供的配置内容和配置方式远不止上面这些，它提供了非常丰富和灵活的配置方法，下面我们将详解介绍 HystrixPropertiesStrategy 实现的各项配置属性。在具体说明这些属性之前，我们需要了解一下这些属性都存在下面 4 个不同优先级别的配置（优先级由低到高）。

- **全局默认值**：如果没有设置下面三个级别的属性，那么这个属性就是默认值。由于该属性通过代码定义，所以对于这个级别，我们主要关注它在代码中定义的默认值即可。
- **全局配置属性**：通过在配置文件中定义全局属性值，在应用启动时或在与 Spring Cloud Config 和 Spring Cloud Bus 实现的动态刷新配置功能配合下，可以实现对“全局默认值”的覆盖以及在运行期对“全局默认值”的动态调整。
- **实例默认值**：通过代码为实例定义的默认值。通过代码的方式为实例设置属性值来覆盖默认的全局配置。

- **实例配置属性**：通过配置文件来为指定的实例进行属性配置，以覆盖前面的三个默认值。它也可用 Spring Cloud Config 和 Spring Cloud Bus 实现的动态刷新配置功能实现对具体实例配置的动态调整。

通过理解 Hystrix 4 个级别的属性配置，对设置 Hystrix 的默认值以及在线上如何根据实际情况去调整配置非常有帮助，下面我们来具体看看它有哪些具体的属性配置。

Command 属性

Command 属性主要用来控制 HystrixCommand 命令的行为。

它主要有下面 5 种不同类型的属性配置。

execution 配置

execution 配置控制的是 HystrixCommand.run()的执行。

- execution.isolation.strategy：该属性用来设置 HystrixCommand.run()执行的隔离策略，它有如下两个选项。
  - THREAD：通过线程池隔离的策略。它在独立的线程上执行，并且它的并发限制受线程池中线程数量的限制。
  - SEMAPHORE：通过信号量隔离的策略。它在调用线程上执行，并且它的并发限制受信号量计数的限制。

属性级别	默认值、配置方式、配置属性
全局默认值	THREAD
全局配置属性	hystrix.command.default.execution.isolation.strategy
实例默认值	通过 HystrixCommandProperties.Setter().withExecutionIsolationStrategy (ExecutionIsolationStrategy.THREAD)设置，也可通过@HystrixProperty (name="execution.isolation.strategy", value= "THREAD")注解设置
实例配置属性	hystrix.command.HystrixCommandKey.execution.isolation.strategy

- execution.isolation.thread.timeoutInMilliseconds：该属性用来配置 HystrixCommand 执行的超时时间，单位为毫秒。当 HystrixCommand 执行时间超过该配置值之后，Hystrix 会将该执行命令标记为 TIMEOUT 并进入服务降级处理逻辑。

属性级别	默认值、配置方式、配置属性
全局默认值	1000 毫秒

续表

属性级别	默认值、配置方式、配置属性
全局配置属性	hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds
实例默认值	通过 HystrixCommandProperties.Setter().withExecutionTimeoutInMilliseconds(int value) 设置，也可通过 @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",value="2000") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.execution.isolation.thread.timeoutInMilliseconds

- execution.timeout.enabled：该属性用来配置 HystrixCommand.run() 的执行是否启用超时时间。默认为 true，如果设置为 false，那么 execution.isolation.thread.timeoutInMilliseconds 属性的配置将不再起作用。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.execution.timeout.enabled
实例默认值	通过 HystrixCommandProperties.Setter().withExecutionTimeoutEnabled(boolean value) 设置，也可通过 @HystrixProperty(name="execution.timeout.enabled", value="false") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.execution.timeout.enabled

- execution.isolation.thread.interruptOnTimeout：该属性用来配置当 HystrixCommand.run() 执行超时的时候是否要将它中断。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.execution.isolation.thread.interruptOnTimeout
实例默认值	通过 HystrixCommandProperties.Setter().withExecutionIsolationThreadInterruptOnTimeout(boolean value) 设置，也可通过 @HystrixProperty(name="execution.isolation.thread.interruptOnTimeout", value="false") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.execution.isolation.thread.interruptOnTimeout

- execution.isolation.thread.interruptOnCancel：该属性用来配置当 HystrixCommand.run()执行被取消的时候是否要将它中断。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.execution.isolation.thread.interruptOnCancel
实例默认值	通过 HystrixCommandProperties.Setter().withExecutionIsolationThreadInterruptOnCancel(boolean value)设置，也可通过@HystrixProperty(name="execution.isolation.thread.interruptOnCancel", value="false")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.execution.isolation.thread.interruptOnCancel

- execution.isolation.semaphore.maxConcurrentRequests：当HystrixCommand 的隔离策略使用信号量的时候，该属性用来配置信号量的大小（并发请求数）。当最大并发请求数达到该设置值时，后续的请求将会被拒绝。

属性级别	默认值、配置方式、配置属性
全局默认值	10
全局配置属性	hystrix.command.default.execution.isolation.semaphore.maxConcurrentRequests
实例默认值	通过 HystrixCommandProperties.Setter().withExecutionIsolationSemaphoreMaxConcurrentRequests(int value)设置，也可通过@HystrixProperty(name="execution.isolation.semaphore.maxConcurrentRequests", value="2")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.execution.isolation.semaphore.maxConcurrentRequests

fallback 配置

下面这些属性用来控制 HystrixCommand.getFallback()的执行。这些属性同时适用于线程池的信号量的隔离策略。

- fallback.isolation.semaphore.maxConcurrentRequests：该属性用来设置从调用线程中允许 HystrixCommand.getFallback()方法执行的最大并发请求数。当达到最大并发请求数时，后续的请求将会被拒绝并抛出异常（因为它已经没有后续的 fallback 可以被调用了）。

属性级别	默认值、配置方式、配置属性
全局默认值	10
全局配置属性	hystrix.command.default.fallback.isolation.semaphore.maxConcurrentRequests
实例默认值	通过 HystrixCommandProperties.Setter().withFallbackIsolationSemaphoreMaxConcurrentRequests(int value)设置，也可通过@HystrixProperty(name="fallback.isolation.semaphore.maxConcurrentRequests", value="20")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.fallback.isolation.semaphore.maxConcurrentRequests

- fallback.enabled：该属性用来设置服务降级策略是否启用，如果设置为 false，那么当请求失败或者拒绝发生时，将不会调用 HystrixCommand.getFallback() 来执行服务降级逻辑。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.fallback.enabled
实例默认值	通过 HystrixCommandProperties.Setter().withFallbackEnabled(boolean value)设置，也可通过@HystrixProperty(name= "fallback.enabled", value="false")注解来设置
实例配置属性	配置属性：hystrix.command.HystrixCommandKey.fallback.enabled

circuitBreaker 配置

下面这些是断路器的属性配置，用来控制 HystrixCircuitBreaker 的行为。

- circuitBreaker.enabled：该属性用来确定当服务请求命令失败时，是否使用断路器来跟踪其健康指标和熔断请求。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.circuitBreaker.enabled
实例默认值	通过 HystrixCommandProperties.Setter().withCircuitBreakerEnabled(boolean value)设置，也可通过@HystrixProperty(name="circuitBreaker.enabled",value="false")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.circuitBreaker.enabled

- circuitBreaker.requestVolumeThreshold：该属性用来设置在滚动时间窗中，断路器熔断的最小请求数。例如，默认该值为 20 的时候，如果滚动时间窗（默认 10 秒）内仅收到了 19 个请求，即使这 19 个请求都失败了，断路器也不会打开。

属性级别	默认值、配置方式、配置属性
全局默认值	20
全局配置属性	hystrix.command.default.circuitBreaker.requestVolumeThreshold
实例默认值	通过 HystrixCommandProperties.Setter().withCircuitBreakerRequest-VolumeThreshold(int value) 设置，也可通过@HystrixProperty(name="circuitBreaker.requestVolumeThreshold", value="30") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.circuitBreaker.requestVolumeThreshold

- circuitBreaker.sleepWindowInMilliseconds：该属性用来设置当断路器打开之后的休眠时间窗。休眠时间窗结束之后，会将断路器置为“半开”状态，尝试熔断的请求命令，如果依然失败就将断路器继续设置为“打开”状态，如果成功就设置为“关闭”状态。

属性级别	默认值、配置方式、配置属性
全局默认值	5000
全局配置属性	hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds
实例默认值	通过 HystrixCommandProperties.Setter().withCircuitBreakerSleep-WindowInMilliseconds(int value) 设置，也可通过@HystrixProperty(name="circuitBreaker.sleepWindowInMilliseconds",value="3000") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.circuitBreaker.sleepWindowInMilliseconds

- circuitBreaker.errorThresholdPercentage：该属性用来设置断路器打开的错误百分比条件。例如，默认值为 5000 的情况下，表示在滚动时间窗中，在请求数量超过 circuitBreaker.requestVolumeThreshold 阈值的前提下，如果错误请求数的百分比超过 50，就把断路器设置为“打开”状态，否则就设置为“关闭”状态。

属性级别	默认值、配置方式、配置属性
全局默认值	50

续表

属性级别	默认值、配置方式、配置属性
全局配置属性	hystrix.command.default.circuitBreaker.errorThresholdPercentage
实例默认值	通过 HystrixCommandProperties.Setter().withCircuitBreakerError-ThresholdPercentage(int value) 设置，也可通过 @HystrixProperty (name="circuitBreaker.errorThresholdPercentage", value="40") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.circuitBreaker.errorThresholdPercentage

- circuitBreaker.forceOpen：如果将该属性设置为 true，断路器将强制进入“打开”状态，它会拒绝所有请求。该属性优先于 circuitBreaker.forceClosed 属性。

属性级别	默认值、配置方式、配置属性
全局默认值	false
全局配置属性	hystrix.command.default.circuitBreaker.forceOpen
实例默认值	通过 HystrixCommandProperties.Setter().withCircuitBreakerForceOpen (boolean value) 设置，也可通过@HystrixProperty (name="circuitBreaker.forceOpen", value="true") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.circuitBreaker. forceOpen

- circuitBreaker.forceClosed：如果将该属性设置为 true，断路器将强制进入“关闭”状态，它会接收所有请求。如果 circuitBreaker.forceOpen 属性为 true，该属性不会生效。

属性级别	默认值、配置方式、配置属性
全局默认值	false
全局配置属性	hystrix.command.default.circuitBreaker.forceClosed
实例默认值	通过 HystrixCommandProperties.Setter().withCircuitBreakerForce-Closed(boolean value) 设置，也可通过 @HystrixProperty(name= "circuitBreaker.forceClosed", value="true") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.circuitBreaker. forceClosed

metrics 配置

下面的属性均与 HystrixCommand 和 HystrixObservableCommand 执行中捕获

的指标信息有关。

- metrics.rollingStats.timeInMilliseconds：该属性用来设置滚动时间窗的长度，单位为毫秒。该时间用于断路器判断健康度时需要收集信息的持续时间。断路器在收集指标信息的时候会根据设置的时间窗长度拆分成多个“桶”来累计各度量值，每个“桶”记录了一段时间内的采集指标。例如，当采用默认值 10000 毫秒时，断路器默认将其拆分成 10 个桶（桶的数量也可通过 metrics.rollingStats.numBuckets 参数设置），每个桶记录 1000 毫秒内的指标信息。

属性级别	默认值、配置方式、配置属性
全局默认值	10000
全局配置属性	hystrix.command.default.metrics.rollingStats.timeInMilliseconds
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsRollingStatisticalWindowInMilliseconds(int value) 设置，也可通过 @HystrixProperty(name="metrics.rollingStats.timeInMilliseconds", value="20000")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.metrics.rollingStats.timeInMilliseconds

注意：该属性从 Hystrix 1.4.12 版本开始（Brixton.SR5 版本中使用了 1.5.3 版本），只有在应用初始化的时候生效，通过动态刷新配置不会产生效果，这样做是为了避免出现运行期监测数据丢失的情况。

- metrics.rollingStats.numBuckets：该属性用来设置滚动时间窗统计指标信息时划分“桶”的数量。

属性级别	默认值、配置方式、配置属性
全局默认值	10
全局配置属性	hystrix.command.default.metrics.rollingStats.numBuckets
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsRollingStatisticalWindowBuckets(int value) 设置，也可通过 @HystrixProperty(name="metrics.rollingStats.numBuckets",value="20")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.metrics.rollingStats.numBuckets

注意：metrics.rollingStats.timeInMilliseconds 参数的设置必须能够被 metrics.rollingStats.numBuckets 参数整除，不然将抛出异常。该参数与 metrics.rollingStats.timeInMilliseconds 一样，从 Hystrix 1.4.12 版本开始（Brixton.SR5 版本中使用了 1.5.3 版本），只有在应用初始化的时候生效，通过动态刷新配置不会产生效果，这样做是为了避免出现运行期监测数据丢失的情况。

- metrics.rollingPercentile.enabled：该属性用来设置对命令执行的延迟是否使用百分位数来跟踪和计算。如果设置为 false，那么所有的概要统计都将返回 -1。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.metrics.rollingPercentile.enabled
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsRollingPercentileEnabled(boolean value) 设置，也可通过 @HystrixProperty(name="metrics.rollingPercentile.enabled", value="false") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.metrics.rollingPercentile.enabled

- metrics.rollingPercentile.timeInMilliseconds：该属性用来设置百分位统计的滚动窗口的持续时间，单位为毫秒。

属性级别	默认值、配置方式、配置属性
全局默认值	60000
全局配置属性	hystrix.command.default.metrics.rollingPercentile.timeInMilliseconds
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsRollingPercentileWindowInMilliseconds(int value) 设置，也可通过 @HystrixProperty(name="metrics.rollingPercentile.timeInMilliseconds", value="50000") 注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.metrics.rollingPercentile.timeInMilliseconds

> 注意：该属性从 Hystrix 1.4.12 版本开始（Brixton.SR5 版本中使用了 1.5.3 版本），只有在应用初始化的时候生效，通过动态刷新配置不会产生效果，这样做是为了避免出现运行期监测数据丢失的情况。

- metrics.rollingPercentile.numBuckets：该属性用来设置百分位统计滚动窗口中使用“桶”的数量。

属性级别	默认值、配置方式、配置属性
全局默认值	6
全局配置属性	hystrix.command.default.metrics.rollingPercentile.numBuckets
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsRollingPercentileWindowBuckets(int value)设置，也可通过@HystrixProperty(name="metrics.rollingPercentile.numBuckets",value="5")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.metrics.rollingPercentile.numBuckets

> 注意：metrics.rollingPercentile.timeInMilliseconds 参数的设置必须能够被 metrics.rollingPercentile.numBuckets 参数整除，不然将会抛出异常。该属性从 Hystrix 1.4.12 版本开始（Brixton.SR5 版本中使用了 1.5.3 版本），只有在应用初始化的时候生效，通过动态刷新配置不会产生效果，这样做是为了避免出现运行期监测数据丢失的情况。

- metrics.rollingPercentile.bucketSize：该属性用来设置在执行过程中每个“桶”中保留的最大执行次数。如果在滚动时间窗内发生超过该设定值的执行次数，就从最初的位置开始重写。例如，将该值设置为 100，滚动窗口为 10 秒，若在 10 秒内一个“桶”中发生了 500 次执行，那么该“桶”中只保留最后的 100 次执行的统计。另外，增加该值的大小将会增加内存量的消耗，并增加排序百分位数所需的计算时间。

属性级别	默认值、配置方式、配置属性
全局默认值	100
全局配置属性	hystrix.command.default.metrics.rollingPercentile.bucketSize
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsRollingPercentileBucketSize(int value)设置，也可通过@HystrixProperty(name="metrics.rollingPercentile.bucketSize",value= "120")注解来设置

续表

属性级别	默认值、配置方式、配置属性
实例配置属性	hystrix.command.HystrixCommandKey.metrics.rollingPercentile.bucketSize

注意：该属性从 Hystrix 1.4.12 版本开始（Brixton.SR5 版本中使用了 1.5.3 版本），只有在应用初始化的时候生效，通过动态刷新配置不会产生效果，这样做是为了避免出现运行期监测数据丢失的情况。

- metrics.healthSnapshot.intervalInMilliseconds：该属性用来设置采集影响断路器状态的健康快照（请求的成功、错误百分比）的间隔等待时间。

属性级别	默认值、配置方式、配置属性
全局默认值	500
全局配置属性	hystrix.command.default.metrics.healthSnapshot.intervalInMilliseconds
实例默认值	通过 HystrixCommandProperties.Setter().withMetricsHealthSnapshotIntervalInMilliseconds(int value)设置，也可通过@HystrixProperty(name="metrics.healthSnapshot.intervalInMilliseconds",value="600")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.metrics.healthSnapshot.intervalInMilliseconds

requestContext 配置

下面这些属性涉及 HystrixCommand 使用的 HystrixRequestContext 的设置。

- requestCache.enabled：此属性用来配置是否开启请求缓存。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.requestCache.enabled
实例默认值	通过 HystrixCommandProperties.Setter().withRequestCacheEnabled(boolean value)设置，也可通过@HystrixProperty(name="requestCache.enabled", value="false")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.requestCache.enabled

- requestLog.enabled：该属性用来设置 HystrixCommand 的执行和事件是否打印日志到 HystrixRequestLog 中。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.command.default.requestLog.enabled
实例默认值	通过 HystrixCommandProperties.Setter().withRequestLogEnabled(boolean value)设置，也可通过@HystrixProperty(name="requestLog.enabled", value="false")注解来设置
实例配置属性	hystrix.command.HystrixCommandKey.requestLog.enabled

collapser 属性

该属性除了在代码中用 set 和配置文件配置之外，也可使用注解进行配置。可使用 @ HystrixCollapser 中的 collapserProperties 属性来设置，比如：

```
@HystrixCollapser(batchMethod = "batch", collapserProperties = {
    @HystrixProperty(name="timerDelayInMilliseconds", value = "20")
})
```

下面这些属性用来控制命令合并相关的行为。

- maxRequestsInBatch：该参数用来设置一次请求合并批处理中允许的最大请求数。

属性级别	默认值、配置方式、配置属性
全局默认值	Integer.MAX_VALUE
全局配置属性	hystrix.collapser.default.maxRequestsInBatch
实例默认值	通过 HystrixCollapserProperties.Setter().withMaxRequestsInBatch(int value)设置，也可通过@HystrixProperty(name="maxRequestsInBatch", value="false")注解来设置
实例配置属性	hystrix.collapser.HystrixCollapserKey.maxRequestsInBatch

- timerDelayInMilliseconds：该参数用来设置批处理过程中每个命令延迟的时间，单位为毫秒。

属性级别	默认值、配置方式、配置属性
全局默认值	10
全局配置属性	hystrix.collapser.default.timerDelayInMilliseconds
实例默认值	通过 HystrixCollapserProperties.Setter().withTimerDelayIn-Milliseconds(int value)设置，也可通过@HystrixProperty(name="timerDelayInMilliseconds",value="20")注解来设置
实例配置属性	hystrix.collapser.HystrixCollapserKey.timerDelayInMilliseconds

- requestCache.enabled：该参数用来设置批处理过程中是否开启请求缓存。

属性级别	默认值、配置方式、配置属性
全局默认值	true
全局配置属性	hystrix.collapser.default.requestCache.enabled
实例默认值	通过 HystrixCollapserProperties.Setter().withRequestCacheEnabled(boolean value)设置，也可通过@HystrixProperty(name="requestCache.enabled", value="false")注解来设置
实例配置属性	hystrix.collapser.HystrixCollapserKey.requestCache. enabled

threadPool 属性

该属性除了在代码中用 set 和配置文件配置之外，还可使用注解进行配置。可使用@HystrixCommand 中的 threadPoolProperties 属性来设置，比如：

```
@HystrixCommand(fallbackMethod = "helloFallback", commandKey = "helloKey",
    threadPoolProperties = {
        @HystrixProperty(name="coreSize", value = "20")
    }
)
```

下面这些属性用来控制 Hy strix 命令所属线程池的配置。

- coreSize：该参数用来设置执行命令线程池的核心线程数，该值也就是命令执行的最大并发量。

属性级别	默认值、配置方式、配置属性
全局默认值	10
全局配置属性	hystrix.threadpool.default.coreSize
实例默认值	通过 HystrixThreadPoolProperties.Setter().withCoreSize(int value)设置，也可通过@HystrixProperty(name="coreSize", value="false")注解来设置
实例配置属性	hystrix.threadpool.HystrixThreadPoolKey.coreSize

- maxQueueSize：该参数用来设置线程池的最大队列大小。当设置为-1 时，线程池将使用 SynchronousQueue 实现的队列，否则将使用 LinkedBlockingQueue 实现的队列。

属性级别	默认值、配置方式、配置属性
全局默认值	-1
全局配置属性	hystrix.threadpool.default.maxQueueSize

续表

属性级别	默认值、配置方式、配置属性
实例默认值	通过 HystrixThreadPoolProperties.Setter().withMaxQueueSize (int value) 设置，也可通过 @HystrixProperty(name="maxQueueSize", value="10")注解来设置
实例配置属性	hystrix.threadpool.HystrixThreadPoolKey.maxQueueSize

注意：该属性只有在初始化的时候才有用，无法通过动态刷新的方式来调整。

- queueSizeRejectionThreshold：该参数用来为队列设置拒绝阈值。通过该参数，即使队列没有达到最大值也能拒绝请求。该参数主要是对 LinkedBlocking-Queue 队列的补充，因为 LinkedBlockingQueue 队列不能动态修改它的对象大小，而通过该属性就可以调整拒绝请求的队列大小了。

属性级别	默认值、配置方式、配置属性
全局默认值	5
全局配置属性	hystrix.threadpool.default.queueSizeRejectionThreshold
实例默认值	通过 HystrixThreadPoolProperties.Setter().withQueueSizeRejection-Threshold(int value) 设置，也可通过 @HystrixProperty(name="queueSizeRejectionThreshold", value="10")注解来设置
实例配置属性	hystrix.threadpool.HystrixThreadPoolKey.queueSizeRejection-Threshold

注意：当 maxQueueSize 属性为-1 的时候，该属性不会生效。

- metrics.rollingStats.timeInMilliseconds：该参数用来设置滚动时间窗的长度，单位为毫秒。该滚动时间窗的长度用于线程池的指标度量，它会被分成多个“桶”来统计指标。

属性级别	默认值、配置方式、配置属性
全局默认值	10000
全局配置属性	hystrix.threadpool.default.metrics.rollingStats.time-InMilliseconds
实例默认值	通过 HystrixThreadPoolProperties.Setter().withMetricsRolling-StatisticalWindowInMilliseconds(int value)设置，也可通过@HystrixProperty(name="metrics.rollingStats.timeInMilliseconds", value="10")注解来设置

续表

属性级别	默认值、配置方式、配置属性
实例配置属性	hystrix.threadpool.HystrixThreadPoolKey.metrics.rollingStats.timeInMilliseconds

- metrics.rollingStats.numBuckets：该参数用来设置滚动时间窗被划分成“桶”的数量。

属性级别	默认值、配置方式、配置属性
全局默认值	10
全局配置属性	hystrix.threadpool.default.metrics.rollingStats. numBuckets
实例默认值	通过 HystrixThreadPoolProperties.Setter().withMetricsRollingStatisticalWindowBuckets(int value) 设置，也可通过@HystrixProperty(name="metrics.rollingStats.numBuckets", value="10")注解来设置
实例配置属性	hystrix.threadpool.HystrixThreadPoolKey.metrics.rollingStats.numBuckets

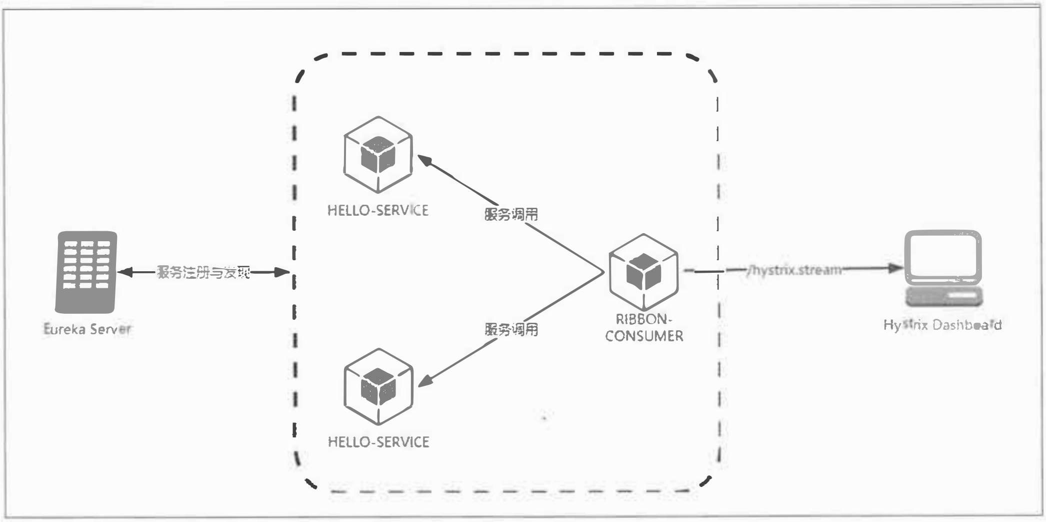
注意： metrics.rollingStats.timeInMilliseconds 参数的设置必须能够被 metrics.rolling-Stats.numBuckets 参数整除，不然将会抛出异常。

Hystrix 仪表盘

在断路器原理的介绍中，我们多次提到关于请求命令的度量指标的判断。这些度量指标都是 HystrixCommand 和 HystrixObservableCommand 实例在执行过程中记录的重要信息，它们除了在 Hystrix 断路器实现中使用之外，对于系统运维也有非常大的帮助。这些指标信息会以“滚动时间窗”与“桶”结合的方式进行汇总，并在内存中驻留一段时间，以供内部或外部进行查询使用，Hystrix 仪表盘就是这些指标内容的消费者之一。

通过之前的内容，我们已经体验到了 Spring Cloud 对 Hystrix 的优雅整合。除此之外，Spring Cloud 还完美地整合了它的仪表盘组件 Hystrix Dashboard，它主要用来实时监控 Hystrix 的各项指标信息。通过 Hystrix Dashboard 反馈的实时信息，可以帮助我们快速发现系统中存在的问题，从而及时地采取应对措施。

本节中我们将在 Hystrix 入门例子的基础上，构建一个 Hystrix Dashboard 来对 RIBBON-CONSUMER 实现监控，完成后的架构如下图所示。



在 Spring Cloud 中构建一个 Hystrix Dashboard 非常简单，只需要下面 4 步：

- 创建一个标准的 Spring Boot 工程，命名为 hystrix-dashboard。
- 编辑 pom.xml，具体依赖内容如下所示：

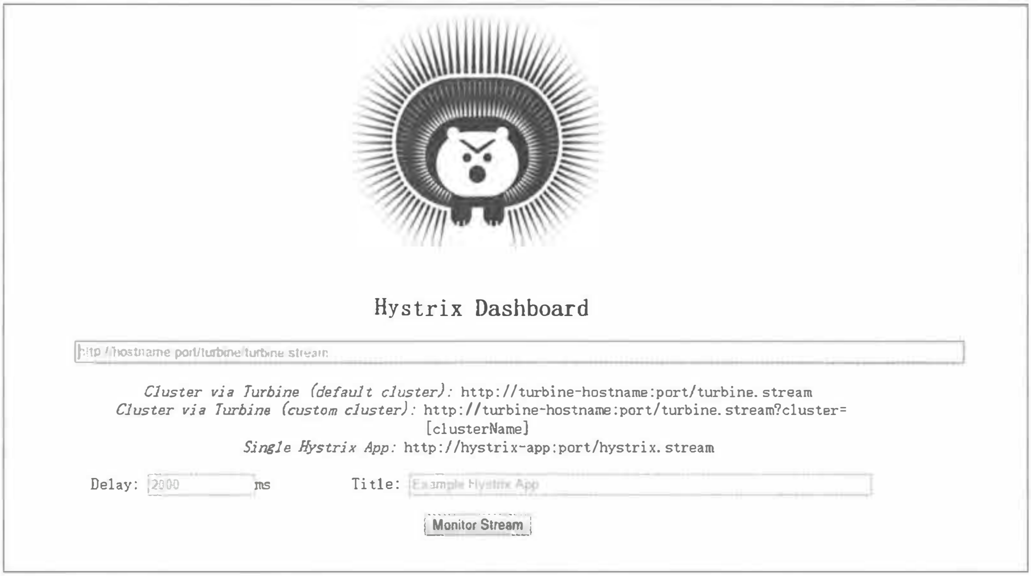
```
<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>Brixton.SR5</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

- 为应用主类加上@EnableHystrixDashboard，启用 Hystrix Dashboard 功能。

- 根据实际情况修改 application.properties 配置文件，比如选择一个未被占用的端口等，此步不是必需的。

```
spring.application.name=hystrix-dashboard
server.port=2001
```

到这里我们已经完成了基本配置，接下来可以启动该应用，并访问 http://localhost:2001/hystrix。可以看到如下页面：



这是 Hystrix Dashboard 的监控首页，该页面中并没有具体的监控信息。从页面的文字内容中我们可以知道，Hystrix Dashboard 共支持三种不同的监控方式，如下所示。

- **默认的集群监控**：通过 URL http://turbine-hostname:port/turbine.stream 开启，实现对默认集群的监控。
- **指定的集群监控**：通过 URL http://turbine-hostname:port/turbine.stream?cluster=[clusterName] 开启，实现对 clusterName 集群的监控。
- **单体应用的监控**：通过 URL http://hystrix-app:port/hystrix.stream 开启，实现对具体某个服务实例的监控。

前两者都是对集群的监控，需要整合 Turbine 才能实现，这部分内容我们将在下一节中做详细介绍。在本节中，我们主要实现对单个服务实例的监控，这里我们先来实现单个服务实例的监控。

既然 Hystrix Dashboard 监控单实例节点需要通过访问实例的/hystrix.stream 接口来实现，我们自然需要为服务实例添加这个端点，而添加该功能的步骤也同样简单，只需

要下面两步。

- 在服务实例pom.xml中的dependencies节点中新增spring-boot-starter-actuator 监控模块以开启监控相关的端点，并确保已经引入断路器的依赖spring-cloud-starter-hystrix：

```
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-hystrix</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

- 确保在服务实例的主类中已经使用@EnableCircuitBreaker 注解，开启了断路器功能。

在为 RIBBON-CONSUMER 加入上面的配置之后，重启它的实例，此时我们可以在控制台中看到打印了大量的监控端点，其中/hystrix.stream 就是用于 Hystrix Dashboard 来展现监控信息的接口。

```
    o.s.b.a.e.mvc.EndpointHandlerMapping    : Mapped "{[/hystrix.stream/**]}" onto
public org.springframework.web.servlet.ModelAndView
org.springframework.cloud.netflix.endpoint.ServletWrappingEndpoint.handle(javax.ser
vlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse) throws
java.lang.Exception
```

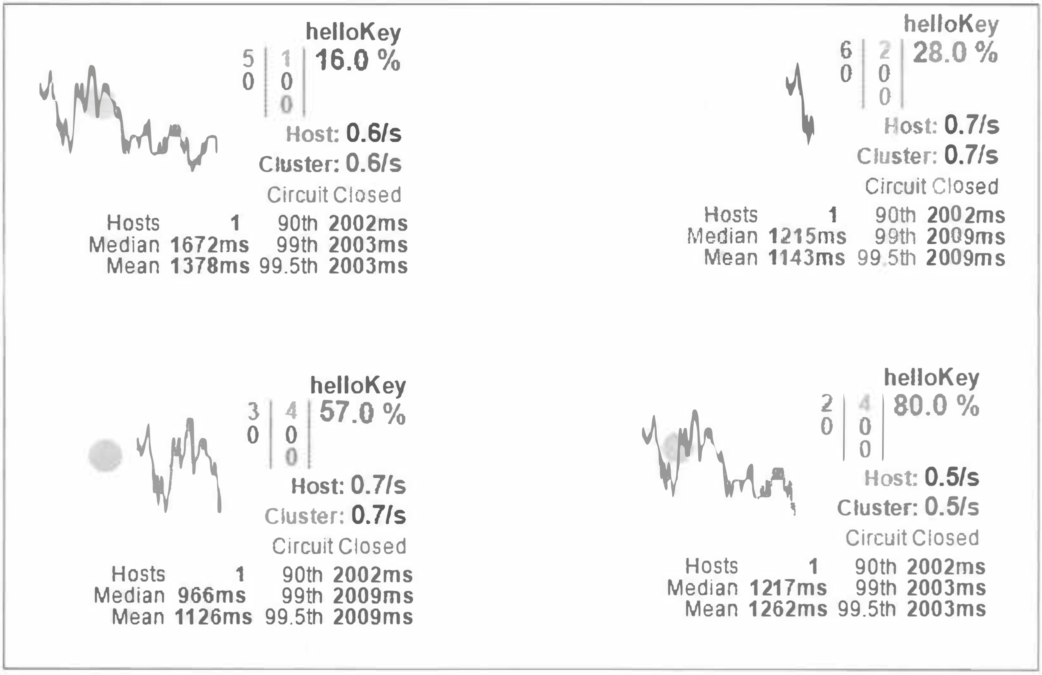
到这里已经完成了所有的配置，在 Hystrix Dashboard 的首页输入 http://localhost:9000/hystrix.stream，可以看到已启动对 RIBBON-CONSUMER 的监控，单击 Monitor Stream 按钮，可以看到如下页面。

在对该页面进行介绍之前，先看看在首页中我们还没有介绍的另外两个参数。

- `Delay`：该参数用来控制服务器上轮询监控信息的延迟时间，默认为 2000 毫秒，可以通过配置该属性来降低客户端的网络和 CPU 消耗。
- `Title`：该参数对应了上图头部标题 Hystrix Stream 之后的内容，默认会使用具体监控实例的 URL，可以通过配置该信息来展示更合适的标题。

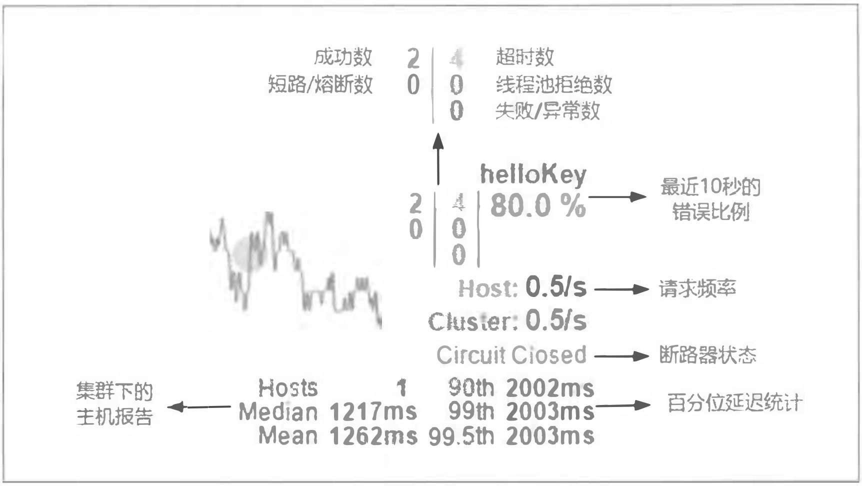
回到监控页面，我们来详细说说其中各元素的具体含义。

- 可以在监控信息的左上部找到两个重要的图形信息：一个实心圆和一条曲线。
  - 实心圆：其有两种含义。通过颜色的变化代表了实例的健康程度，如下图所示，它的健康度从绿色、黄色、橙色、红色递减。该实心圆除了颜色的变化之外，它的大小也会根据实例的请求流量发生变化，流量越大该实心圆就越大。所以通过该实心圆的展示，我们可以在大量的实例中快速发现故障实例和高压力实例。



  - 曲线：用来记录 2 分钟内流量的相对变化，可以通过它来观察流量的上升和下降趋势。

- 其他一些数量指标如下图所示。



通过本节内容我们已经能够使用 Hystrix Dashboard 来对单个实例做信息监控了，但是在分布式系统中，往往有非常多的实例需要去维护和监控。到目前为止，我们能做的就是通过开启多个窗口来监控多个实例，很显然这样的做法并不合理。在下一节中，我们将介绍利用 Turbine 和 Hystrix Dashboard 配合实现对集群的监控。

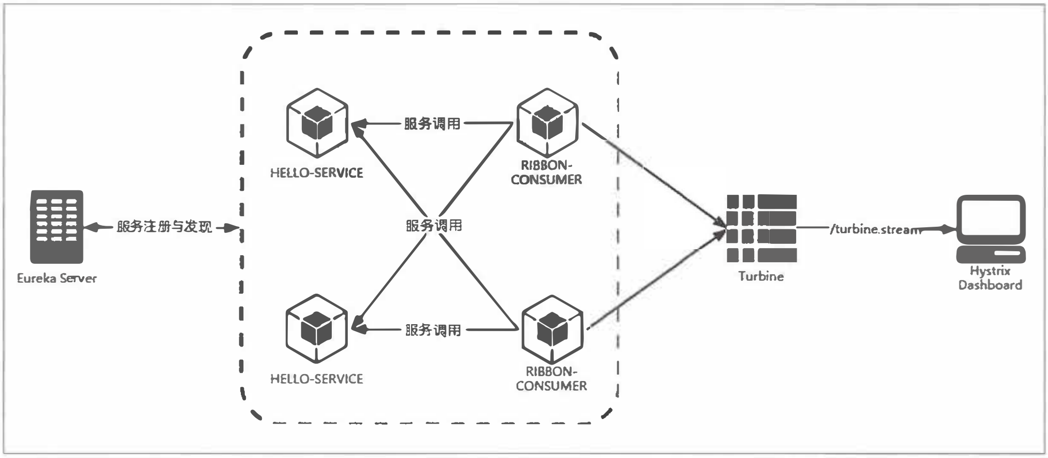
注意：当使用 Hystrix Board 来监控 Spring Cloud Zuul 构建的 API 网关时，Thread Pool 信息会一直处于 Loading 状态。这是由于 Zuul 默认会使用信号量来实现隔离，只有通过 Hystrix 配置把隔离机制改为线程池的方式才能够得以展示。

Turbine 集群监控

在上一节介绍 Hystrix Dashboard 的首页时，我们提到过除了可以开启单个实例的监控页面之外，还有一个监控端点`/turbine.stream`是对集群使用的。从端点的命名中，可猜测到这里我们将引入 Turbine，通过它来汇集监控信息，并将聚合后的信息提供给 Hystrix Dashboard 来集中展示和监控。

构建监控聚合服务

下面我们将在上一节内容的基础上做一些扩展，通过引入 Turbine 来聚合 `RIBBON-CONSUMER` 服务的监控信息，并输出给 Hystrix Dashboard 来进行展示，最后完成如下图所示的结构。



具体实现步骤如下：

- 创建一个标准的 Spring Boot 工程，命名为 turbine。
- 编辑 pom.xml，具体依赖内容如下所示。

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>Brixton.SR5</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-turbine</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

- 创建应用主类 TurbineApplication，并使用@EnableTurbine 注解开启 Turbine。

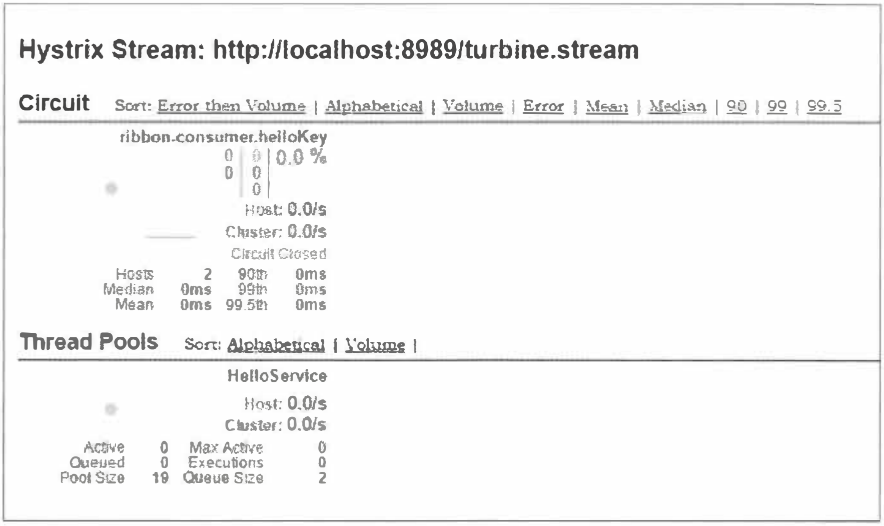
```
@EnableTurbine
@EnableDiscoveryClient
@SpringBootApplication
public class TurbineApplication {

    public static void main(String[] args) {
        SpringApplication.run(TurbineApplication.class, args);
    }

}
```

在 application.properties 中加入 Eureka 和 Turbine 的相关配置，具体如下：

```
spring.application.name=turbine

server.port=8989
management.port=8990

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/

turbine.app-config=RIBBON-CONSUMER
turbine.cluster-name-expression="default"
turbine.combine-host-port=true
```

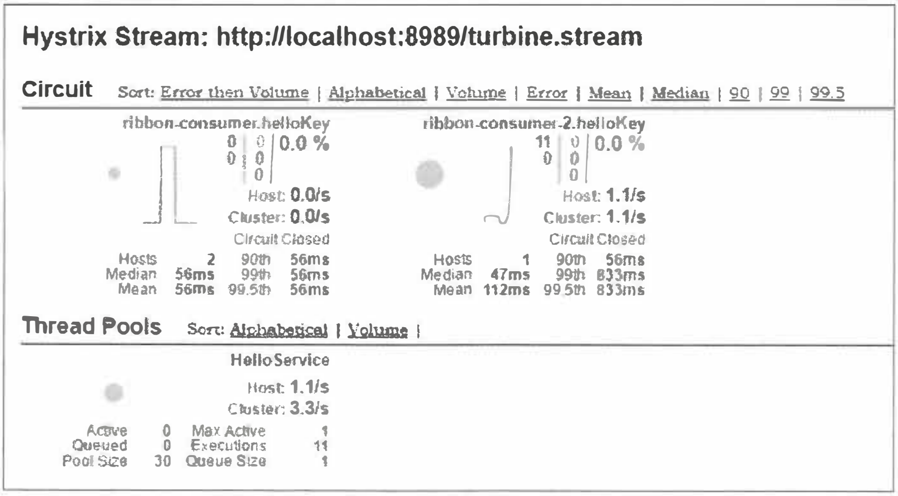
其中，turbine.app-config 参数指定了需要收集监控信息的服务名；turbine.cluster-name-expression 参数指定了集群名称为 default，当服务数量非常多的时候，可以启动多个 Turbine 服务来构建不同的聚合集群，而该参数可以用来区分这些不同的聚合集群，同时该参数值可以在 Hystrix 仪表盘中用来定位不同的聚合集群，只需在 Hystrix Stream 的 URL 中通过 cluster 参数来指定；turbine.combine-host-port 参数设置为 true，可以让同一主机上的服务通过主机名与端口号的组合来进行区分，默认情况下会以 host 来区分不同的服务，这会使得在本地调试的时候，本机上的不同服务聚合成一个服务来统计。

在完成了上面的内容构建之后，我们来体验一下 Turbine 对集群的监控能力。分别启动 eureka-server、HELLO-SERVICE、RIBBON-CONSUMER、Turbine 以及 Hystrix Dashboard。访问 Hystrix Dashboard，并开启对 http://localhost:8989/turbine.stream 的监控，我们可以看到如下页面：



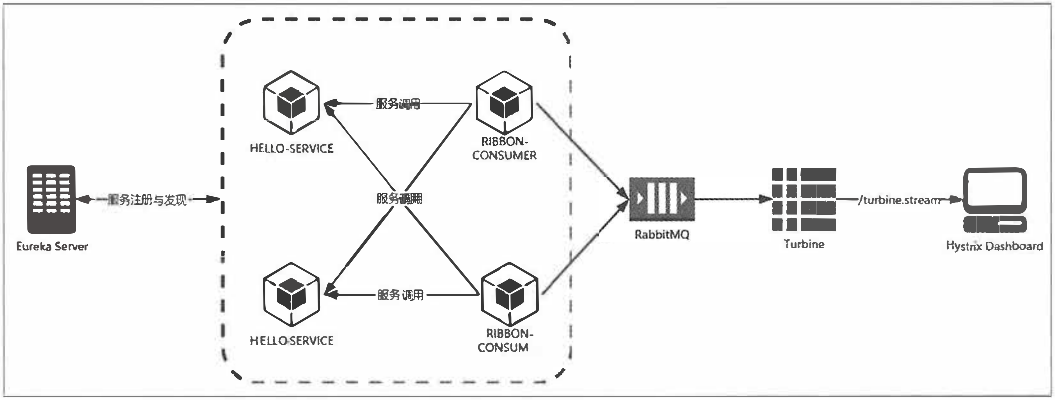
从图中可以看到，虽然我们如之前的架构那样启动了两个 RIBBON-CONSUMER，但是在监控页面中依然只是展示了一个监控图。不过仔细的读者可能已经发现，图中集群报告区域中的 Hosts 属性与之前尝试单机监控时已经有所不同。由此我们可以知道 RIBBON-CONSUMER 启动了两个实例，这里只展现了一个监控图，是由于这两个实例是同一个服务，而对于集群来说我们关注的是服务集群的高可用性，所以 Turbine 会将相同服务作为整体来看待，并汇总成一个监控图。

也可以为 RIBBON-CONSUMER 设置一个新的 spring.application.name，比如 ribbon-consumer-2，并启动它，此时我们就有两个服务会将监控信息输出给 Turbine 汇总了。刷新之前的监控页面，可以看到如下有两个监控图的页面：



与消息代理结合

Spring Cloud 在封装 Turbine 的时候，还封装了基于消息代理的收集实现。所以，我们可以将所有需要收集的监控信息都输出到消息代理中，然后 Turbine 服务再从消息代理中异步获取这些监控信息，最后将这些监控信息聚合并输出到 Hystrix Dashboard 中。通过引入消息代理，我们的 Turbine 和 Hystrix Dashboard 实现的监控架构可以改成如下图所示的结构。



从图中可以看到，这里多了一个重要元素 RabbitMQ。对于 RabbitMQ 的安装我们可以查看第 9 章的相关内容，这里不做过多说明。下面，我们来构建一个新的应用以实现基于消息代理的 Turbine 聚合服务，具体步骤如下所示。

- 创建一个标准的 Spring Boot 工程，命名为 `turbine-amqp`。
- 编辑 `pom.xml`，具体依赖内容如下所示：

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

<parent>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-parent</artifactId>
    <version>Brixton.SR5</version>
    <relativePath /> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-turbine-amqp</artifactId>
```

```
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
</dependencies>
```

可以看到这里主要引入了 spring-cloud-starter-turbine-amqp 依赖，它实际上包装了 spring-cloud-starter-turbine-stream 和 pring-cloud-starter-stream-rabbit。

注意：这里我们需要使用 Java 8 来运行。

- 在应用主类中使用@EnableTurbineStream 注解来启用 Turbine Stream 的配置。

```
@EnableTurbineStream
@EnableDiscoveryClient
@SpringBootApplication
public class TurbineApplication {

    public static void main(String[] args) {
        SpringApplication.run(TurbineApplication.class, args);
    }

}
```

- 配置 application.properties 文件。

```
spring.application.name=turbine

server.port=8989
management.port=8990

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

对于 Turbine 的配置已经完成了，下面需要对服务消费者 RIBBON-CONSUMER 做一些修改，使其监控信息能够输出到 RabbitMQ 上。这个修改也非常简单，只需在 pom.xml 中增加对 spring-cloud-netflix-hystrix-amqp 的依赖，具体如下：

```
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-netflix-hystrix-amqp</artifactId>
    </dependency>
</dependencies>
```

在完成了上面的配置之后，继续启动 eureka-server、HELLO-SERVICE、RIBBON-CONSUMER、Turbine 以及 Hystrix Dashboard，同时确保 RabbitMQ 已在正常运行。访问 Hystrix Dashboard，并开启对 http://localhost:8989/turbine.stream 的监控，我们可以获得如之前实现的同样结果，只是这里的监控信息收集是通过消息代理异步实现的。

第 6 章

声明式服务调用：Spring Cloud Feign

通过前两章对 Spring Cloud Ribbon 和 Spring Cloud Hystrix 的介绍，我们已经掌握了开发微服务应用时的两个重磅武器，学会了如何在微服务架构中实现客户端负载均衡的服务调用以及如何通过断路器来保护我们的微服务应用。这两者将被作为基础工具类框架广泛地应用在各个微服务的实现中，不仅包括我们自身的业务类微服务，也包括一些基础设施类微服务（比如网关）。此外，在实践过程中，我们会发现对这两个框架的使用几乎是同时出现的。既然如此，那么是否有更高层次的封装来整合这两个基础工具以简化开发呢？本章我们即将介绍的 Spring Cloud Feign 就是这样一个工具。它基于 Netflix Feign 实现，整合了 Spring Cloud Ribbon 与 Spring Cloud Hystrix，除了提供这两者的强大功能之外，它还提供了一种声明式的 Web 服务客户端定义方式。

我们在使用 Spring Cloud Ribbon 时，通常都会利用它对 RestTemplate 的请求拦截来实现对依赖服务的接口调用，而 RestTemplate 已经实现了对 HTTP 请求的封装处理，形成了一套模板化的调用方法。在之前的例子中，我们只是简单介绍了 RestTemplate 调用的实现，但是在实际开发中，由于对服务依赖的调用可能不止于一处，往往一个接口会被多处调用，所以我们通常都会针对各个微服务自行封装一些客户端类来包装这些依赖服务的调用。这个时候我们会发现，由于 RestTemplate 的封装，几乎每一个调用都是简单的模板化内容。综合上述这些情况，Spring Cloud Feign 在此基础上做了进一步封装，由它来帮助我们定义和实现依赖服务接口的定义。在 Spring Cloud Feign 的实现下，我们只需创建一个接口并用注解的方式来配置它，即可完成对服务提供方的接口绑定，简化了在使用 Spring Cloud Ribbon 时自行封装服务调用客户端的开发量。Spring Cloud Feign 具备可插拔的注解支持，包括 Feign 注解和 JAX-RS 注解。同时，为了适应 Spring 的广大用户，它在 Netflix Feign

的基础上扩展了对 Spring MVC 的注解支持。这对于习惯于 Spring MVC 的开发者来说，无疑是一个好消息，因为这样可以大大减少学习使用它的成本。另外，对于 Feign 自身的一些主要组件，比如编码器和解码器等，它也以可插拔的方式提供，在有需求的时候我们可以方便地扩展和替换它们。

快速入门

在本节中，我们将通过一个简单的示例来展现 Spring Cloud Feign 在服务客户端定义上所带来的便利。下面的示例将继续使用之前我们实现的 `hello-service` 服务，这里我们会通过 Spring Cloud Feign 提供的声明式服务绑定功能来实现对该服务接口的调用。

- 首先，创建一个 Spring Boot 基础工程，取名为 `feign-consumer`，并在 `pom.xml` 中引入 `spring-cloud-starter-eureka` 和 `spring-cloud-starter-feign` 依赖。具体内容如下所示：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-feign</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
```

```
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- 创建应用主类 ConsumerApplication，并通过@EnableFeignClients 注解开启 Spring Cloud Feign 的支持功能。

```
@EnableFeignClients
@EnableDiscoveryClient
@SpringBootApplication
public class ConsumerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }

}
```

- 定义 HelloService 接口，通过@FeignClient 注解指定服务名来绑定服务，然后再使用 Spring MVC 的注解来绑定具体该服务提供的 REST 接口。

```
@FeignClient("hello-service")
public interface HelloService {

    @RequestMapping("/hello")
    String hello();

}
```

注意：这里服务名不区分大小写，所以使用 hello-service 和 HELLO-SERVICE 都是可以的。另外，在 Brixton.SR5 版本中，原有的 serviceId 属性已经被废弃，若要写属性名，可以使用 name 或 value。

- 接着，创建一个 ConsumerController 来实现对 Feign 客户端的调用。使用@Autowired 直接注入上面定义的 HelloService 实例，并在 helloConsumer 函数中调用这个绑定了 hello-service 服务接口的客户端来向该服务发起/hello 接口的调用。

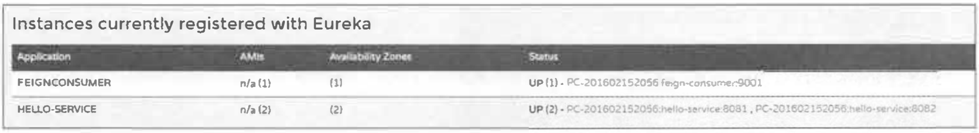
```
@RestController
public class ConsumerController {

    @Autowired
    HelloService helloService;

    @RequestMapping(value = "/feign-consumer", method = RequestMethod.GET)
    public String helloConsumer() {
        return helloService.hello();
    }

}
```

- 最后，同 Ribbon 实现的服务消费者一样，需要在 application.properties 中指定服务注册中心，并定义自身的服务名为 feign-consumer，为了方便本地调试与之前的 Ribbon 消费者区分，端口使用 9001。

```
spring.application.name=feign-consumer
server.port=9001

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

测试验证

如之前验证 Ribbon 客户端负载均衡一样，我们先启动服务注册中心以及两个 HELLO-SERVICE，然后启动 FEIGN-CONSUMER，此时我们在 Eureka 信息面板中可看到如下内容：



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
FEIGNCONSUMER	n/a (1)	(1)	UP (1) - PC-201602152056:feign-consumer:9001
HELLO-SERVICE	n/a (2)	(2)	UP (2) - PC-201602152056:hello-service:8081 , PC-201602152056:hello-service:8082

发送几次 GET 请求到 http://localhost:9001/feign-consumer，可以得到如之前 Ribbon 实现时一样的效果，正确返回了“Hello World”。并且根据控制台的输出，我们可以看到 Feign 实现的消费者，依然是利用 Ribbon 维护了针对 HELLO-SERVICE 的服务列表信息，并且通过轮询实现了客户端负载均衡。而与 Ribbon 不同的是，通过 Feign 我们只需定义服务绑定接口，以声明式的方法，优雅而简单地实现了服务调用。

参数绑定

在上一节的示例中，我们使用 Spring Cloud Feign 实现的是一个不带参数的 REST 服务绑定。然而现实系统中的各种业务接口要比它复杂得多，我们会在 HTTP 的各个位置传入各种不同类型的参数，并且在返回请求响应的时候也可能是一个复杂的对象结构。在本节

中，我们将详细介绍 Feign 中对几种不同形式参数的绑定方法。

在开始介绍 Spring Cloud Feign 的参数绑定之前，我们先扩展一下服务提供方 hello-service。增加下面这些接口定义，其中包含带有 Request 参数的请求、带有 Header 信息的请求、带有 RequestBody 的请求以及请求响应体中是一个对象的请求。

```
@RequestMapping(value = "/hello1", method = RequestMethod.GET)
public String hello(@RequestParam String name) {
    return "Hello " + name;
}

@RequestMapping(value = "/hello2", method = RequestMethod.GET)
public User hello(@RequestHeader String name, @RequestHeader Integer age) {
    return new User(name, age);
}

@RequestMapping(value = "/hello3", method = RequestMethod.POST)
public String hello(@RequestBody User user) {
    return "Hello "+ user.getName() + ", " + user.getAge();
}
```

User 对象的定义如下，这里省略了 getter 和 setter 函数，需要注意的是，这里必须要有 User 的默认构造函数。不然，Spring Cloud Feign 根据 JSON 字符串转换 User 对象时会抛出异常。

```
public class User {

    private String name;
    private Integer age;

    public User() {
    }

    public User(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    // 省略 getter 和 setter

    @Override
    public String toString() {
        return "name=" + name + ", age=" +age;
    }

}
```

在完成了对 hello-service 的改造之后，下面我们开始在快速入门示例的 feign-consumer 应用中实现这些新增的请求的绑定。

- 首先，在 feign-consumer 中创建与上面一样的 User 类。
- 然后，在 HelloService 接口中增加对上述三个新增接口的绑定声明，修改后，完成的 HelloService 接口如下所示：

```
@FeignClient("HELLO-SERVICE")
public interface HelloService {

    @RequestMapping("/hello")
    String hello();

    @RequestMapping(value = "/hello1", method = RequestMethod.GET)
    String hello(@RequestParam("name") String name) ;

    @RequestMapping(value = "/hello2", method = RequestMethod.GET)
    User hello(@RequestHeader("name") String name, @RequestHeader("age") Integer
age);

    @RequestMapping(value = "/hello3", method = RequestMethod.POST)
    String hello(@RequestBody User user);

}
```

这里一定要注意，在定义各参数绑定时，@RequestParam、@RequestHeader 等可以指定参数名称的注解，它们的 value 千万不能少。在 Spring MVC 程序中，这些注解会根据参数名来作为默认值，但是在 Feign 中绑定参数必须通过 value 属性来指明具体的参数名，不然会抛出 IllegalStateException 异常，value 属性不能为空。

```
Caused by: java.lang.IllegalStateException: RequestParam.value() was empty on
parameter 0
    at feign.Util.checkState(Util.java:128)
```

- 最后，在 ConsumerController 中新增一个/feign-consumer2 接口，来对本节新增的声明接口进行调用，修改后的完整代码如下所示：

```
@RestController
public class ConsumerController {

    @Autowired
    HelloService helloService;

    @RequestMapping(value = "/feign-consumer", method = RequestMethod.GET)
    public String helloConsumer() {
        return helloService.hello();
```

```
    }

    @RequestMapping(value = "/feign-consumer2", method = RequestMethod.GET)
    public String helloConsumer2() {
        StringBuilder sb = new StringBuilder();
        sb.append(helloService.hello()).append("\n");
        sb.append(helloService.hello("DIDI")).append("\n");
        sb.append(helloService.hello("DIDI", 30)).append("\n");
        sb.append(helloService.hello(new User("DIDI", 30))).append("\n");
        return sb.toString();
    }
}
```

测试验证

在完成上述改造之后，启动服务注册中心、两个 hello-service 服务以及我们改造过的 feign-consumer。通过发送 GET 请求到 http://localhost:9001/feign-consumer2，触发 HelloService 对新增接口的调用。最终，我们会获得如下输出，代表接口绑定和调用成功。

```
Hello World
Hello DIDI
name=DIDI, age=30
Hello DIDI, 30
```

继承特性

通过“快速入门”以及“参数绑定”小节中的示例实践，相信很多读者已经观察到，当使用 Spring MVC 的注解来绑定服务接口时，我们几乎完全可以从服务提供方的 Controller 中依靠复制操作，构建出相应的服务客户端绑定接口。既然存在这么多复制操作，我们自然需要考虑这部分内容是否可以得到进一步的抽象呢？在 Spring Cloud Feign 中，针对该问题提供了继承特性来帮助我们解决这些复制操作，以进一步减少编码量。下面，我们详细看看如何通过 Spring Cloud Feign 的继承特性来实现 REST 接口定义的复用。

- 为了能够复用 DTO 与接口定义，我们先创建一个基础的 Maven 工程，命名为 hello-service-api。
- 由于在 hello-service-api 中需要定义可同时复用于服务端与客户端的接口，我们要使用到 Spring MVC 的注解，所以在 pom.xml 中引入 spring-boot-starter-web 依赖，具体内容如下所示：

```
<groupId>com.didispace</groupId>
<artifactId>hello-service-api</artifactId>
```

```
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>hello-service-api</name>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.3.7.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
    </dependencies>
```

- 将上一节中实现的 User 对象复制到 hello-service-api 工程中，比如保存到 com.didispace.dto.User。
- 在 hello-service-api 工程中创建 com.didispace.service.HelloService 接口，内容如下，该接口中的 User 对象为本项目中的 com.didispace.dto.User。
- 将上一节中实现的 User 对象复制到 com.didispace.dto.User。创建 com.didispace.service.HelloService 接口，内容如下，该接口中的 User 对象为本项目中的 com.didispace.dto.User。

```
    @RequestMapping("/refactor")
    public interface HelloService {

        @RequestMapping(value = "/hello4", method = RequestMethod.GET)
        String hello(@RequestParam("name") String name) ;

        @RequestMapping(value = "/hello5", method = RequestMethod.GET)
        User hello(@RequestHeader("name") String name, @RequestHeader("age") Integer
age);
```

```
    @RequestMapping(value = "/hello6", method = RequestMethod.POST)
    String hello(@RequestBody User user);

}
```

因为后续还会通过之前的 hello-service 和 feign-consumer 来重构，所以为了避免接口混淆，在这里定义 HelloService 时，除了头部定义了/rafactor 前缀之外，同时将提供服务的三个接口更名为/hello4、/hello5、/hello6。

- 下面对 hello-service 进行重构，在 pom.xml 的 dependency 节点中，新增对 hello-service-api 的依赖。

```
<dependency>
    <groupId>com.didispace</groupId>
    <artifactId>hello-service-api</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

- 创建 RefactorHelloController 类继承 hello-service-api 中定义的 HelloService 接口，并参考之前的 HelloController 来实现这三个接口，具体内容如下所示：

```
@RestController
public class RefactorHelloController implements HelloService {

    @Override
    public String hello(@RequestParam("name") String name) {
        return "Hello " + name;
    }

    @Override
    public User hello(@RequestHeader("name") String name, @RequestHeader("age")
Integer age) {
        return new User(name, age);
    }

    @Override
    public String hello(@RequestBody User user) {
        return "Hello "+ user.getName() + ", " + user.getAge();
    }

}
```

我们可以看到通过继承的方式，在 Controller 中不再包含以往会定义的请求映射注解 @RequestMapping，而参数的注解定义在重写的时候会自动带过来。在这个类中，除了

要实现接口逻辑之外，只需再增加@RestController 注解使该类成为一个 REST 接口类就大功告成了。

- 完成了服务提供者的重构，接下来在服务消费者 feign-consumer 的 pom.xml 文件中，如在服务提供者中一样，新增对 hello-service-api 的依赖。
- 创建 RefactorHelloService 接口，并继承 hello-service-api 包中的 HelloService 接口，然后添加@FeignClient 注解来绑定服务。

```
@FeignClient(value = "HELLO-SERVICE")
public interface RefactorHelloService extends com.didispace.service.HelloService {

}
```

- 最后，在 ConsumerController 中，注入 RefactorHelloService 的实例，并新增一个请求/feign-consumer3 来触发对 RefactorHelloService 的实例的调用。

```
@Autowired
RefactorHelloService refactorHelloService;

@RequestMapping(value = "/feign-consumer3", method = RequestMethod.GET)
public String helloConsumer3() {
    StringBuilder sb = new StringBuilder();
    sb.append(refactorHelloService.hello("MIMI")).append("\n");
    sb.append(refactorHelloService.hello("MIMI", 20)).append("\n");
    sb.append(refactorHelloService.hello(new com.didispace.dto.User("MIMI",
20))).append("\n");
    return sb.toString();
}
```

测试验证

这次的验证过程需要注意几个工程的构建顺序，由于 hello-service 和 feign-consumer 都依赖 hello-service-api 工程中的接口和 DTO 定义，所以必须先构建hello-service-api 工程，然后再构建hello-service 和 feign-consumer。接着我们分别启动服务注册中心，hello-service 和 feign-consumer，并访问 http://localhost:9001/feign-consumer3，调用成功后可以获得如下输出：

```
Hello MIMI
name=MIMI, age=20
Hello MIMI, 20
```

优点与缺点

使用 Spring Cloud Feign 继承特性的优点很明显，可以将接口的定义从 Controller 中剥

离，同时配合 Maven 私有仓库就可以轻易地实现接口定义的共享，实现在构建期的接口绑定，从而有效减少服务客户端的绑定配置。这么做虽然可以很方便地实现接口定义和依赖的共享，不用再复制粘贴接口进行绑定，但是这样的做法使用不当的话会带来副作用。由于接口在构建期间就建立起了依赖，那么接口变动就会对项目构建造成影响，可能服务提供方修改了一个接口定义，那么会直接导致客户端工程的构建失败。所以，如果开发团队通过此方法来实现接口共享的话，建议在开发评审期间严格遵守面向对象的开闭原则，尽可能地做好前后版本的兼容，防止牵一发而动全身的后果，增加团队不必要的维护工作量。

Ribbon 配置

由于 Spring Cloud Feign 的客户端负载均衡是通过 Spring Cloud Ribbon 实现的，所以我们可以直接通过配置 Ribbon 客户端的方式来自定义各个服务客户端调用的参数。那么我们如何在使用 Spring Cloud Feign 的工程中使用 Ribbon 的配置呢？

全局配置

全局配置的方法非常简单，我们可以直接使用 `ribbon.<key>=<value>`的方式来设置 ribbon 的各项默认参数。比如，修改默认的客户端调用超时时间：

```
ribbon.ConnectTimeout=500
ribbon.ReadTimeout=5000
```

指定服务配置

大多数情况下，我们对于服务调用的超时时间可能会根据实际服务的特性做一些调整，所以仅仅依靠默认的全局配置是不行的。在使用 Spring Cloud Feign 的时候，针对各个服务客户端进行个性化配置的方式与使用 Spring Cloud Ribbon 时的配置方式是一样的，都采用 `<client>.ribbon.key=value` 的格式进行设置。但是，这里就有一个疑问了，`<client>`所指代的 Ribbon 客户端在哪里呢？

回想一下，在定义 Feign 客户端的时候，我们使用了`@FeignClient` 注解。在初始化过程中，Spring Cloud Feign 会根据该注解的 `name` 属性或 `value` 属性指定的服务名，自动创建一个同名的 Ribbon 客户端。也就是说，在之前的示例中，使用`@FeignClient(value = "HELLO-SERVICE")`来创建 Feign 客户端的时候，同时也创建了一个名为 `HELLO-SERVICE` 的 Ribbon 客户端。既然如此，我们就可以使用`@FeignClient` 注解中的 `name` 或 `value` 属性值来设置对应的 Ribbon 参数，比如：

```
HELLO-SERVICE.ribbon.ConnectTimeout=500
HELLO-SERVICE.ribbon.ReadTimeout=2000
HELLO-SERVICE.ribbon.OkToRetryOnAllOperations=true
```

```
HELLO-SERVICE.ribbon.MaxAutoRetriesNextServer=2
HELLO-SERVICE.ribbon.MaxAutoRetries=1
```

重试机制

在 Spring Cloud Feign 中默认实现了请求的重试机制，而上面我们对于 HELLO-SERVICE 客户端的配置内容就是对于请求超时以及重试机制配置的详情，具体内容可参考第 4 章最后一节关于 Spring Cloud Ribbon 重试机制的介绍。我们可以通过修改之前的示例做一些验证。

- 在 hello-service 应用的/hello 接口实现中，增加一些随机延迟，比如：

```
@RequestMapping(value = "/hello", method = RequestMethod.GET)
public String hello() throws Exception {
    ServiceInstance instance = client.getLocalServiceInstance()     ;
    // 测试超时
    int sleepTime = new Random().nextInt(3000);
    logger.info("sleepTime:" + sleepTime);
    Thread.sleep(sleepTime);

    logger.info("/hello, host:" + instance.getHost() + ", service_id:" +
instance.getServiceId());
    return "Hello World";
}
```

- 在 feign-consumer 应用中增加上文中提到的重试配置参数。其中，由于 HELLO-SERVICE.ribbon.MaxAutoRetries 设置为 1，所以重试策略先尝试访问首选实例一次，失败后才更换实例访问，而更换实例访问的次数通过 HELLO-SERVICE.ribbon.MaxAutoRetriesNextServer 参数设置为 2，所以会尝试更换两次实例进行重试。
- 最后，启动这些应用，并尝试访问几次 http://localhost:9001/feign-consumer 接口。当请求发生超时的时候，我们在 hello-service 的控制台中可能会获得如下输出内容（由于 sleepTime 的随机性，并不一定每次相同）：

```
INFO 19264 --- [nio-8001-exec-6] com.didispace.web.HelloController     : sleepTime:2929
INFO 19264 --- [nio-8001-exec-7] com.didispace.web.HelloController   : sleepTime:253
INFO 19264 --- [nio-8001-exec-7] com.didispace.web.HelloController   : /hello, host:
192.168.0.105, service_id:hello-service
INFO 19264 --- [nio-8001-exec-6] com.didispace.web.HelloController   : /hello, host:
192.168.0.105, service_id:hello-service
```

从控制台输出中，我们可以看到这次访问的第一次请求延迟时间为 2929 毫秒，由于超时时间设置为 2000 毫秒，Feign 客户端发起了重试，第二次请求的延迟为 253 秒，

没有超时。Feign 客户端在进行服务调用时，虽然经历了一次失败，但是通过重试机制，最终还是获得了请求结果。所以，对于重试机制的实现，对于构建高可用的服务集群来说非常重要，而 Spring Cloud Feign 也为其提供了足够的支持。

这里需要注意一点，Ribbon 的超时与 Hystrix 的超时是两个概念。为了让上述实现有效，我们需要让 Hystrix 的超时时间大于 Ribbon 的超时时间，否则 Hystrix 命令超时后，该命令直接熔断，重试机制就没有任何意义了。

Hystrix 配置

在 Spring Cloud Feign 中，除了引入了用于客户端负载均衡的 Spring Cloud Ribbon 之外，还引入了服务保护与容错的工具 Hystrix。默认情况下，Spring Cloud Feign 会为将所有 Feign 客户端的方法都封装到 Hystrix 命令中进行服务保护。在上一节末尾，我们介绍重试机制的配置时，也提到了关于 Hystrix 的超时时间配置。那么在本节中，我们就来详细介绍一下，如何在使用 Spring Cloud Feign 时配置 Hystrix 属性以及如何实现服务降级。

全局配置

对于 Hystrix 的全局配置同 Spring Cloud Ribbon 的全局配置一样，直接使用它的默认配置前缀 `hystrix.command.default` 就可以进行设置，比如设置全局的超时时间：

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=5000
```

另外，在对 Hystrix 进行配置之前，我们需要确认 `feign.hystrix.enabled` 参数没有被设置为 `false`，否则该参数设置会关闭 Feign 客户端的 Hystrix 支持。而对于我们之前测试重试机制时，对于 Hystrix 的超时时间控制除了可以使用上面的配置来增加熔断超时时间，也可以通过 `feign.hystrix.enabled=false` 来关闭 Hystrix 功能，或者使用 `hystrix.command.default.execution.timeout.enabled=false` 来关闭熔断功能。

禁用 Hystrix

上文我们提到了，在 Spring Cloud Feign 中，可以通过 `feign.hystrix.enabled=false` 来关闭 Hystrix 功能。另外，如果不想全局地关闭 Hystrix 支持，而只想针对某个服务客户端关闭 Hystrix 支持时，需要通过使用`@Scope("prototype")`注解为指定的客户端配置 `Feign.Builder` 实例，详细实现步骤如下所示。

- 构建一个关闭 Hystrix 的配置类。

```
@Configuration
```

```
public class DisableHystrixConfiguration {

    @Bean
    @Scope("prototype")
    public Feign.Builder feignBuilder() {
        return Feign.builder();
    }

}
```

- 在 HelloService 的@FeignClient 注解中，通过 configuration 参数引入上面实现的配置。

```
@FeignClient(name="HELLO-SERVICE", configuration = DisableHystrixConfiguration.class)
public interface HelloService {
    ...
}
```

指定命令配置

对于 Hystrix 命令的配置，在实际应用时往往也会根据实际业务情况制定出不同的配置方案。配置方法也跟传统的 Hystrix 命令的参数配置相似，采用 hystrix.command.<commandKey>作为前缀。而<commandKey>默认情况下会采用 Feign 客户端中的方法名作为标识，所以，针对上一节介绍的尝试机制中对/hello 接口的熔断超时时间的配置可以通过其方法名作为<commandKey>来进行配置，具体如下：

```
hystrix.command.hello.execution.isolation.thread.timeoutInMilliseconds=5000
```

在使用指定命令配置的时候，需要注意，由于方法名很有可能重复，这个时候相同方法名的 Hystrix 配置会共用，所以在进行方法定义与配置的时候需要做好一定的规划。当然，也可以重写 Feign.Builder 的实现，并在应用主类中创建它的实例来覆盖自动化配置的 HystrixFeign.Builder 实现。

服务降级配置

Hystrix 提供的服务降级是服务容错的重要功能，由于 Spring Cloud Feign 在定义服务客户端的时候与 Spring Cloud Ribbon 有很大差别，HystrixCommand 定义被封装了起来，我们无法像之前介绍 Spring Cloud Hystrix 时，通过@HystrixCommand注解的 fallback 参数那样来指定具体的服务降级处理方法。但是，Spring Cloud Feign 提供了另外一种简单的定义方式，下面我们在之前创建的 feign-consumer 工程中进行改造。

- 服务降级逻辑的实现只需要为Feign客户端的定义接口编写一个具体的接口实现类。比如为 HelloService 接口实现一个服务降级类 HelloServiceFallback，其中每个重写方法的实现逻辑都可以用来定义相应的服务降级逻辑，具体如下：

```
@Component
public class HelloServiceFallback implements HelloService {

    @Override
    public String hello() {
        return "error";
    }

    @Override
    public String hello(@RequestParam("name") String name) {
        return "error";
    }

    @Override
    public User hello(@RequestHeader("name") String name, @RequestHeader("age")
Integer age) {
        return new User("未知", 0);
    }

    @Override
    public String hello(@RequestBody User user) {
        return "error";
    }

}
```

- 在服务绑定接口 HelloService 中，通过@FeignClient 注解的 fallback 属性来指定对应的服务降级实现类。

```
@FeignClient(name="HELLO-SERVICE", fallback = HelloServiceFallback.class)
public interface HelloService {

    @RequestMapping("/hello")
    String hello();

    @RequestMapping(value = "/hello1", method = RequestMethod.GET)
    String hello(@RequestParam("name") String name) ;

    @RequestMapping(value = "/hello2", method = RequestMethod.GET)
    User hello(@RequestHeader("name") String name, @RequestHeader("age") Integer
age);
```

```
    @RequestMapping(value = "/hello3", method = RequestMethod.POST)
    String hello(@RequestBody User user);

}
```

测试验证

下面我们来验证一下服务降级逻辑的实现。启动服务注册中心和 feign-consumer，但是不启动 hello-service 服务。发送 GET 请求到 http://localhost:9001/feign-consumer2，该接口会分别调用 HelloService 中的 4 个绑定接口，但因为 hello-service 服务没有启动，会直接触发服务降级，并获得下面的输出内容：

```
error
error
name=未知, age=0
error
```

正如我们在 HelloServiceFallback 类中实现的内容，每一个服务接口的断路器实际就是实现类中的重写函数的实现。

> **注意**：在 Brixton.SR5 版本中，fallback 的实现函数中不再支持返回 com.netflix.hystrix.HystrixCommand 和 rx.Observable 类型的异步执行方式和响应式执行方式。

其他配置

请求压缩

Spring Cloud Feign 支持对请求与响应进行 GZIP 压缩，以减少通信过程中的性能损耗。我们只需通过下面两个参数设置，就能开启请求与响应的压缩功能：

```
feign.compression.request.enabled=true
feign.compression.response.enabled=true
```

同时，我们还能对请求压缩做一些更细致的设置，比如下面的配置内容指定了压缩的请求数据类型，并设置了请求压缩的大小下限，只有超过这个大小的请求才会对其进行压缩。

```
feign.compression.request.enabled=true
feign.compression.request.mime-types=text/xml,application/xml,application/json
feign.compression.request.min-request-size=2048
```

上述配置的 feign.compression.request.mime-types 和 feign.compression.request.min-request-size 均为默认值。

日志配置

Spring Cloud Feign 在构建被@FeignClient 注解修饰的服务客户端时，会为每一个客户端都创建一个 feign.Logger 实例，我们可以利用该日志对象的 DEBUG 模式来帮助分析 Feign 的请求细节。可以在 application.properties 文件中使用 logging.level.<FeignClient>的参数配置格式来开启指定 Feign 客户端的 DEBUG 日志，其中<FeignClient>为 Feign 客户端定义接口的完整路径，比如针对本章中我们实现的 HelloService 可以按如下配置开启：

```
logging.level.com.didispace.web.HelloService=DEBUG
```

但是，只是添加了如上配置，还无法实现对 DEBUG 日志的输出。这时由于 Feign 客户端默认的 Logger.Level 对象定义为 NONE 级别，该级别不会记录任何 Feign 调用过程中的信息，所以我们需要调整它的级别，针对全局的日志级别，可以在应用主类中直接加入 Logger.Level 的 Bean 创建，具体如下：

```
@EnableFeignClients
@EnableDiscoveryClient
@SpringBootApplication
public class ConsumerApplication {

    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }

    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }

}
```

当然也可以通过实现配置类，然后在具体的 Feign 客户端来指定配置类以实现是否要调整不同的日志级别，比如下面的实现：

```
@Configuration
public class FullLogConfiguration {
    @Bean
    Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
```

```
}

@FeignClient(name="HELLO-SERVICE", configuration = FullLogConfiguration.class)
public interface HelloService {
    ...
}
```

在调整日志级别为 FULL 之后，我们可以再访问一下之前的 http://localhost:9001/feign-consumer 接口，这时我们在 feign-consumer 的控制台中就可以看到类似下面的请求详细日志：

```
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] ---> GET http://HELLO-SERVICE/hello HTTP/1.1
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] ---> END HTTP (0-byte body)
    ...
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] <--- HTTP/1.1 200 OK (3342ms)
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] Server: Apache-Coyote/1.1
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] Content-Length: 11
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] X-Application-Context: hello-service:8001
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] Date: Sat, 07 Jan 2017 11:49:11 GMT
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] Content-Type: text/plain;charset=UTF-8
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello]
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] Hello World
    DEBUG 20488 --- [HELLO-SERVICE-1] com.didispace.web.HelloService :
[HelloService#hello] <--- END HTTP (11-byte body)
```

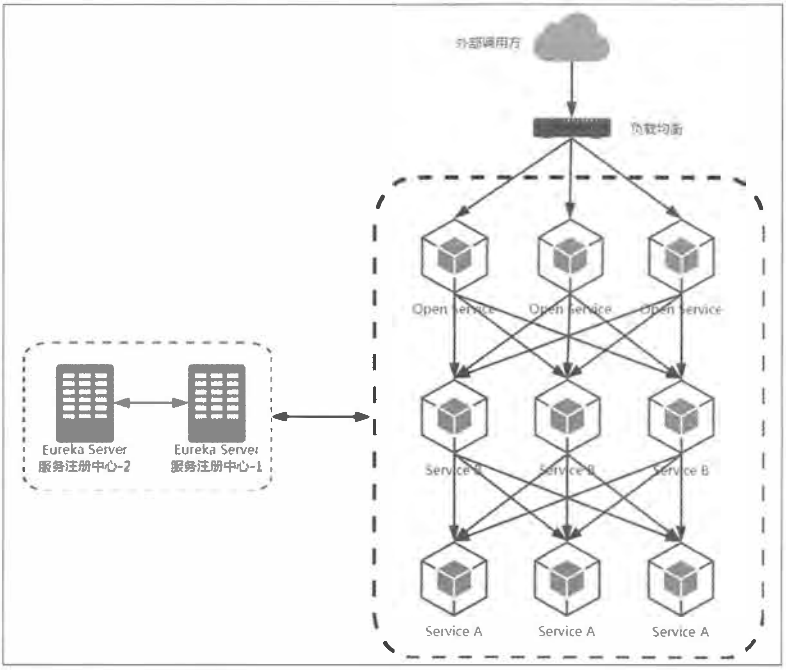
对于 Feign 的 Logger 级别主要有下面 4 类，可根据实际需要进行调整使用。

- NONE：不记录任何信息。
- BASIC：仅记录请求方法、URL 以及响应状态码和执行时间。
- HEADERS：除了记录 BASIC 级别的信息之外，还会记录请求和响应的头信息。
- FULL：记录所有请求与响应的明细，包括头信息、请求体、元数据等。

第 7 章

API 网关服务：Spring Cloud Zuul

通过前几章的介绍，我们对于 Spring Cloud Netflix 下的核心组件已经了解了一大半。这些组件基本涵盖了微服务架构中最为基础的几个核心设施，利用这些组件我们已经可以构建起一个简单的微服务架构系统，比如，通过使用 Spring Cloud Eureka 实现高可用的服务注册中心以及实现微服务的注册与发现；通过 Spring Cloud Ribbon 或 Feign 实现服务间负载均衡的接口调用；同时，为了使分布式系统更为健壮，对于依赖的服务调用使用 Spring Cloud Hystrix 来进行包装，实现线程隔离并加入熔断机制，以避免在微服务架构中因个别服务出现异常而引起级联故障蔓延。通过上述思路，我们可以设计出类似下图的基础系统架构。



在该架构中，我们的服务集群包含内部服务 Service A 和 Service B，它们都会向 Eureka Server 集群进行注册与订阅服务，而 Open Service 是一个对外的 RESTful API 服务，它通过 F5、Nginx 等网络设备或工具软件实现对各个微服务的路由与负载均衡，并公开给外部的客户端调用。

在本章中，我们将把视线聚焦在对外服务这块内容，通常也称为边缘服务。首先需要肯定的是，上面的架构实现系统功能是完全没有问题的，但是我们还是可以进一步思考一下，这样的架构是否还有不足的地方会使运维人员或开发人员感到痛苦。

首先，我们从运维人员的角度来看看，他们平时都需要做一些什么工作来支持这样的架构。当客户端应用单击某个功能的时候往往会发出一些对微服务获取资源的请求到后端，这些请求通过 F5、Nginx 等设施的路由和负载均衡分配后，被转发到各个不同的服务实例上。而为了让这些设施能够正确路由与分发请求，运维人员需要手工维护这些路由规则与服务实例列表，当有实例增减或是 IP 地址变动等情况发生的时候，也需要手工地去同步修改这些信息以保持实例信息与中间件配置内容的一致性。在系统规模不大的时候，维护这些信息的工作还不会太过复杂，但是如果当系统规模不断增大，那么这些看似简单的维护任务会变得越来越难，并且出现配置错误的概率也会逐渐增加。很显然，这样的做法并不可取，所以我们需要一套机制来有效降低维护路由规则与服务实例列表的难度。

其次，我们再从开发人员的角度来看看，在这样的架构下，会产生一些怎样的问题呢？大多数情况下，为了保证对外服务的安全性，我们在服务端实现的微服务接口，往往都会有一定的权限校验机制，比如对用户登录状态的校验等；同时为了防止客户端在发起请求时被篡改等安全方面的考虑，还会有一些签名校验的机制存在。这时候，由于使用了微服务架构的理念，我们将原本处于一个应用中的多个模块拆成了多个应用，但是这些应用提供的接口都需要这些校验逻辑，我们不得不在这些应用中都实现这样一套校验逻辑。随着微服务规模的扩大，这些校验逻辑的冗余变得越来越多，突然有一天我们发现这套校验逻辑有个 BUG 需要修复，或者需要对其做一些扩展和优化，此时我们就不得不去每个应用里修改这些逻辑，而这样的修改不仅会引起开发人员的抱怨，更会加重测试人员的负担。所以，我们也需要一套机制能够很好地解决微服务架构中，对于微服务接口访问时各前置校验的冗余问题。

为了解决上面这些常见的架构问题，API 网关的概念应运而生。API 网关是一个更为智能的应用服务器，它的定义类似于面向对象设计模式中的 Facade 模式，它的存在就像是整个微服务架构系统的门面一样，所有的外部客户端访问都需要经过它来进行调度和过滤。它除了要实现请求路由、负载均衡、校验过滤等功能之外，还需要更多能力，比如与服务治理框架的结合、请求转发时的熔断机制、服务的聚合等一系列高级功能。

既然 API 网关对于微服务架构这么重要，那么在 Spring Cloud 中是否有相应的解决方

案呢？答案是很肯定的，在 Spring Cloud 中了提供了基于 Netflix Zuul 实现的 API 网关组件——Spring Cloud Zuul。那么，它是如何解决上面这两个普遍问题的呢？

首先，对于路由规则与服务实例的维护问题。Spring Cloud Zuul 通过与 Spring Cloud Eureka 进行整合，将自身注册为 Eureka 服务治理下的应用，同时从 Eureka 中获得了所有其他微服务的实例信息。这样的设计非常巧妙地将服务治理体系中维护的实例信息利用起来，使得将维护服务实例的工作交给了服务治理框架自动完成，不再需要人工介入。而对于路由规则的维护，Zuul 默认会将通过以服务名作为 ContextPath 的方式来创建路由映射，大部分情况下，这样的默认设置已经可以实现我们大部分的路由需求，除了一些特殊情况（比如兼容一些老的 URL）还需要做一些特别的配置。但是相比于之前架构下的运维工作量，通过引入 Spring Cloud Zuul 实现 API 网关后，已经能够大大减少了。

其次，对于类似签名校验、登录校验在微服务架构中的冗余问题。理论上来说，这些校验逻辑在本质上与微服务应用自身的业务并没有多大的关系，所以它们完全可以独立成一个单独的服务存在，只是它们被剥离和独立出来之后，并不是给各个微服务调用，而是在 API 网关服务上进行统一调用来对微服务接口做前置过滤，以实现对微服务接口的拦截和校验。Spring Cloud Zuul 提供了一套过滤器机制，它可以很好地支持这样的任务。开发者可以通过使用 Zuul 来创建各种校验过滤器，然后指定哪些规则的请求需要执行校验逻辑，只有通过校验的才会被路由到具体的微服务接口，不然就返回错误提示。通过这样的改造，各个业务层的微服务应用就不再需要非业务性质的校验逻辑了，这使得我们的微服务应用可以更专注于业务逻辑的开发，同时微服务的自动化测试也变得更容易实现。

微服务架构虽然可以将我们的开发单元拆分得更为细致，有效降低了开发难度，但是它所引出的各种问题如果处理不当会成为实施过程中的不稳定因素，甚至掩盖掉原本实施微服务带来的优势。所以，在微服务架构的实施方案中，API 网关服务的使用几乎成为了必然的选择。

下面我们将详细介绍 Spring Cloud Zuul 的使用方法、配置属性以及一些不足之处和需要进行的思考。

快速入门

介绍了这么多关于 API 网关服务的概念和作用，在这一节中，我们不妨用实际的示例来直观地体验一下 Spring Cloud Zuul 中封装的 API 网关是如何使用和运作，并应用到微服务架构中去的。

构建网关

首先，在实现各种 API 网关服务的高级功能之前，我们需要做一些准备工作，比如，构建起最基本的 API 网关服务，并且搭建几个用于路由和过滤使用的微服务应用等。对于微服务应用，我们可以直接使用之前章节实现的 hello-service 和 feign-consumer。虽然之前我们一直将 feign-consumer 视为消费者，但是在 Eureka 的服务注册与发现体系中，每个服务既是提供者也是消费者，所以 feign-consumer 实质上也是一个服务提供者。之前我们访问的 http://localhost:9001/feign-consumer 等一系列接口就是它提供的服务。读者也可以使用自己实现的微服务应用，因为这部分不是本章的重点，任何微服务应用都可以被用来进行后续的试验。这里，我们详细介绍一下 API 网关服务的构建过程。

- 创建一个基础的 Spring Boot 工程，命名为 api-gateway，并在 pom.xml 中引入 spring-cloud-starter-zuul 依赖，具体如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zuul</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

  对于 spring-cloud-starter-zuul 依赖，可以通过查看它的依赖内容了解到：该模块中不仅包含了 Netflix Zuul 的核心依赖 zuul-core，它还包含了下面这些网关服务需要的重要依赖。

- `spring-cloud-starter-hystrix`：该依赖用来在网关服务中实现对微服务转发时候的保护机制，通过线程隔离和断路器，防止微服务的故障引发 API 网关资源无法释放，从而影响其他应用的对外服务。
- `spring-cloud-starter-ribbon`：该依赖用来实现在网关服务进行路由转发时候的客户端负载均衡以及请求重试。
- `spring-boot-starter-actuator`：该依赖用来提供常规的微服务管理端点。另外，在 Spring Cloud Zuul 中还特别提供了`/routes`端点来返回当前的所有路由规则。

● 创建应用主类，使用`@EnableZuulProxy`注解开启 Zuul 的 API 网关服务功能。

```
@EnableZuulProxy
@SpringCloudApplication
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }
}
```

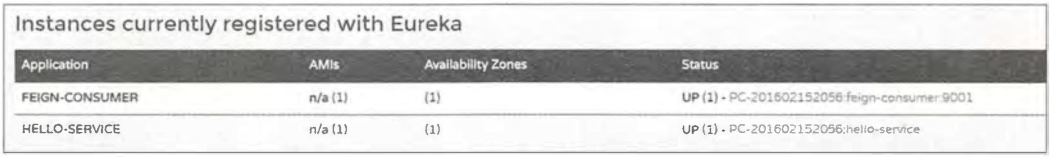
● 在`application.properties`中配置 Zuul 应用的基础信息，如应用名、服务端口号等，具体内容如下：

```
spring.application.name=api-gateway
server.port=5555
```

完成上面的工作后，通过 Zuul 实现的 API 网关服务就构建完毕了。

请求路由

下面，我们将通过一个简单的示例来为上面构建的网关服务增加请求路由的功能。为了演示请求路由的功能，我们先将之前准备的 Eureka 服务注册中心和微服务应用都启动起来。此时，我们在 Eureka 信息面板中可以看到如下图所示的两个微服务应用已经被注册成功了。



传统路由方式

使用 Spring Cloud Zuul 实现路由功能非常简单，只需要对`api-gateway`服务增加一

些关于路由规则的配置，就能实现传统的路由转发功能，比如：

```
zuul.routes.api-a-url.path=/api-a-url/**
zuul.routes.api-a-url.url=http://localhost:8080/
```

该配置定义了发往 API 网关服务的请求中，所有符合/api-a-url/**规则的访问都将被路由转发到 http://localhost:8080/ 地址上，也就是说，当我们访问 http://localhost:5555/api-a-url/hello 的时候，API 网关服务会将该请求路由到 http://localhost:8080/hello 提供的微服务接口上。其中，配置属性 zuul.routes.api-a-url.path 中的 api-a-url 部分为路由的名字，可以任意定义，但是一组 path 和 url 映射关系的路由名要相同，下面将要介绍的面向服务的映射方式也是如此。

面向服务的路由

很显然，传统路由的配置方式对于我们来说并不友好，它同样需要运维人员花费大量的时间来维护各个路由 path 与 url 的关系。为了解决这个问题，Spring Cloud Zuul 实现了与 Spring Cloud Eureka 的无缝整合，我们可以让路由的 path 不是映射具体的 url，而是让它映射到某个具体的服务，而具体的 url 则交给 Eureka 的服务发现机制去自动维护，我们称这类路由为面向服务的路由。在 Zuul 中使用服务路由也同样简单，只需做下面这些配置。

- 为了与 Eureka 整合，我们需要在 api-gateway 的 pom.xml 中引入 spring-cloud-starter-eureka 依赖，具体如下：

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

- 在 api-gateway 的 application.properties 配置文件中指定 Eureka 注册中心的位置，并且配置服务路由。具体如下：

```
zuul.routes.api-a.path=/api-a/**
zuul.routes.api-a.serviceId=hello-service

zuul.routes.api-b.path=/api-b/**
zuul.routes.api-b.serviceId=feign-consumer

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

针对我们之前准备的两个微服务应用 hello-service 和 feign-consumer，在上面的配置中分别定义了两个名为 api-a 和 api-b 的路由来映射它们。另外，通过指定 Eureka Server 服务注册中心的位置，除了将自己注册成服务之外，同时也让 Zuul 能够获取

hello-service 和 feign-consumer 服务的实例清单，以实现 path 映射服务，再从服务中挑选实例来进行请求转发的完整路由机制。

在完成了上面的服务路由配置之后，我们可以将 eureka-server、hello-service、feign-consumer 以及这里用 Spring Clo udZuul 构建的 api-gateway 都启动起来。启动完毕，在 eureka-server 的信息面板中，我们可以看到，除了 hello-service 和 feign-consumer 之外，多了一个网关服务 API-GATEWAY。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - PC-201602152056:api-gateway:5555
FEIGN-CONSUMER	n/a (1)	(1)	UP (1) - PC-201602152056:feign-consumer:9001
HELLO-SERVICE	n/a (1)	(1)	UP (1) - PC-201602152056:hello-service

通过上面的搭建工作，我们已经可以通过服务网关来访问 hello-service 和 feign-consumer 这两个服务了。根据配置的映射关系，分别向网关发起下面这些请求。

- http://localhost:5555/api-a/hello：该 url 符合/api-a/**规则，由 api-a 路由负责转发，该路由映射的 serviceId 为 hello-service，所以最终/hello 请求会被发送到 hello-service 服务的某个实例上去。
- http://localhost:5555/api-b/feign-consumer：该 url 符合/api-b/**规则，由 api-b 路由负责转发，该路由映射的 serviceId 为 feign-consumer，所以最终/feign-consumer 请求会被发送到 feign-consumer 服务的某个实例上去。

通过面向服务的路由配置方式，我们不需要再为各个路由维护微服务应用的具体实例的位置，而是通过简单的 path 与 s ervic eId 的映射组合，使得维护工作变得非常简单。这完全归功于 Spring Clo udEureka 的服务发现机制，它使得 API 网关服务可以自动化完成服务实例清单的维护，完美地解决了对路由映射实例的维护问题。

请求过滤

在实现了请求路由功能之后，我们的微服务应用提供的接口就可以通过统一的 API 网关入口被客户端访问到了。但是，每个客户端用户请求微服务应用提供的接口时，它们的访问权限往往都有一定的限制，系统并不会将所有的微服务接口都对它们开放。然而，目前的服务路由并没有限制权限这样的功能，所有请求都会被毫无保留地转发到具体的应用并返回结果，为了实现对客户端请求的安全校验和权限控制，最简单和粗暴的方法就是为每个微服务应用都实现一套用于校验签名和鉴别权限的过滤器或拦截器。不过，这样的做法并不可取，它会增加日后系统的维护难度，因为同一个系统中的各种校验逻辑很多情况

下都是大致相同或类似的，这样的实现方式会使得相似的校验逻辑代码被分散到了各个微服务中去，冗余代码的出现是我们不希望看到的。所以，比较好的做法是将这些校验逻辑剥离出去，构建出一个独立的鉴权服务。在完成了剥离之后，有不少开发者会直接在微服务应用中通过调用鉴权服务来实现校验，但是这样的做法仅仅只是解决了鉴权逻辑的分离，并没有在本质上将这部分不属于冗余的逻辑从原有的微服务应用中拆分出，冗余的拦截器或过滤器依然会存在。

对于这样的问题，更好的做法是通过前置的网关服务来完成这些非业务性质的校验。由于网关服务的加入，外部客户端访问我们的系统已经有了统一入口，既然这些校验与具体业务无关，那何不在请求到达的时候就完成校验和过滤，而不是转发后再过滤而导致更长的请求延迟。同时，通过在网关中完成校验和过滤，微服务应用端就可以去除各种复杂的过滤器和拦截器了，这使得微服务应用接口的开发和测试复杂度也得到了相应降低。

为了在 API 网关中实现对客户端请求的校验，我们将继续介绍 Spring Cloud Zuul 的另外一个核心功能：请求过滤。Zuul 允许开发者在 API 网关上通过定义过滤器来实现对请求的拦截与过滤，实现的方法非常简单，我们只需要继承 `ZuulFilter` 抽象类并实现它定义的 4 个抽象函数就可以完成对请求的拦截和过滤了。

下面的代码定义了一个简单的 Zuul 过滤器，它实现了在请求被路由之前检查 `HttpServletRequest` 中是否有 `accessToken` 参数，若有就进行路由，若没有就拒绝访问，返回 `401 Unauthorized` 错误。

```
public class AccessFilter extends ZuulFilter {

    private static Logger log = LoggerFactory.getLogger(AccessFilter.class);

    @Override
    public String filterType() {
        return "pre";
    }

    @Override
    public int filterOrder() {
        return 0;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
```

```
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();

        log.info("send {} request to {}", request.getMethod(),
request.getRequestURL().toString());

        Object accessToken = request.getParameter("accessToken");
        if(accessToken == null) {
            log.warn("access token is empty");
            ctx.setSendZuulResponse(false);
            ctx.setResponseStatusCode(401);
            return null;
        }
        log.info("access token ok");
        return null;
    }

}
```

在上面实现的过滤器代码中，我们通过继承 ZuulFilter 抽象类并重写下面 4 个方法来实现自定义的过滤器。这 4 个方法分别定义了如下内容。

- filterType：过滤器的类型，它决定过滤器在请求的哪个生命周期中执行。这里定义为 pre，代表会在请求被路由之前执行。
- filterOrder：过滤器的执行顺序。当请求在一个阶段中存在多个过滤器时，需要根据该方法返回的值来依次执行。
- shouldFilter：判断该过滤器是否需要被执行。这里我们直接返回了 true，因此该过滤器对所有请求都会生效。实际运用中我们可以利用该函数来指定过滤器的有效范围。
- run：过滤器的具体逻辑。这里我们通过 ctx.setSendZuulResponse(false) 令 zuul 过滤该请求，不对其进行路由，然后通过 ctx.setResponseStatusCode(401) 设置了其返回的错误码，当然也可以进一步优化我们的返回，比如，通过 ctx.setResponseBody(body) 对返回的 body 内容进行编辑等。

在实现了自定义过滤器之后，它并不会直接生效，我们还需要为其创建具体的 Bean 才能启动该过滤器，比如，在应用主类中增加如下内容：

```
@EnableZuulProxy
@SpringCloudApplication
public class Application {

    public static void main(String[] args) {
```

```
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

    @Bean
    public AccessFilter accessFilter() {
        return new AccessFilter();
    }
}
```

在对 api-gateway 服务完成了上面的改造之后，我们可以重新启动它，并发起下面的请求，对上面定义的过滤器做一个验证。

- http://localhost:5555/api-a/hello：返回 401 错误。
- http://localhost:5555/api-a/hello&accessToken=token：正确路由到 hello-service 的/hello 接口，并返回 Hello World。

到这里，对于 API 网关服务的快速入门示例就完成了。通过对 Spring Cloud Zuul 两个核心功能的介绍，相信读者已经能够体会到 API 网关服务对微服务架构的重要性了，就目前掌握的 API 网关知识，我们可以将具体原因总结如下：

- 它作为系统的统一入口，屏蔽了系统内部各个微服务的细节。
- 它可以与服务治理框架结合，实现自动化的服务实例维护以及负载均衡的路由转发。
- 它可以实现接口权限校验与微服务业务逻辑的解耦。
- 通过服务网关中的过滤器，在各生命周期中去校验请求的内容，将原本在对外服务层做的校验前移，保证了微服务的无状态性，同时降低了微服务的测试难度，让服务本身更集中关注业务逻辑的处理。

实际上，基于 Spring Cloud Zuul 实现的 API 网关服务除了上面所示的优点之外，它还有一些更加强大的功能，我们将在后续的章节对其进行更深入的介绍。通过本节的内容，我们只是希望以一个简单的例子带领读者先来简单认识一下 API 网关服务提供的基础功能以及它在微服务架构中的重要地位。

路由详解

在“快速入门”一节的请求路由示例中，我们对 Spring Cloud Zuul 中的两类路由功能已经做了简单的使用介绍。在本节中，我们将进一步详细介绍关于 Spring Cloud Zuul 的路由功能，以帮助读者更好地理解和使用它。

传统路由配置

所谓的传统路由配置方式就是在不依赖于服务发现机制的情况下，通过在配置文件中

具体指定每个路由表达式与服务实例的映射关系来实现 API 网关对外部请求的路由。

没有 Eureka 等服务治理框架的帮助，我们需要根据服务实例的数量采用不同方式的配置来实现路由规则。

- **单实例配置**：通过 zuul.routes.<route>.path 与 zuul.routes.<route>.url 参数对的方式进行配置，比如：

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.url=http://localhost:8080/
```

该配置实现了对符合 /user-service/** 规则的请求路径转发到 http://localhost:8080/地址的路由规则。比如，当有一个请求 http://localhost:5555/user-service/hello 被发送到 API 网关上，由于/user-service/hello 能够被上述配置的 path 规则匹配，所以 API 网关会转发请求到 http://localhost:8080/hello 地址。

- **多实例配置**：通过 zuul.routes.<route>.path 与 zuul.routes.<route>.serviceId 参数对的方式进行配置，比如：

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service
ribbon.eureka.enabled=false
user-service.ribbon.listOfServers=http://localhost:8080/,http://localhost:8081/
```

该配置实现了对符合 /user-service/** 规则的请求路径转发到 http://localhost:8080/和 http://localhost:8081/两个实例地址的路由规则。它的配置方式与服务路由的配置方式一样，都采用了 zuul.routes.<route>.path 与 zuul.routes.<route>.serviceId 参数对的映射方式，只是这里的 serviceId 是由用户手工命名的服务名称，配合 ribbon.listOfServers 参数实现服务与实例的维护。由于存在多个实例，API 网关在进行路由转发时需要实现负载均衡策略，于是这里还需要 Spring Cloud Ribbon 的配合。由于在 Spring Cloud Zuul 中自带了对 Ribbon 的依赖，所以我们只需做一些配置即可，比如上面示例中关于 Ribbon 的各个配置，它们的具体作用如下所示。

  - ribbon.eureka.enabled：由于 zuul.routes.<route>.serviceId 指定的是服务名称，默认情况下 Ribbon 会根据服务发现机制来获取配置服务名对应的实例清单。但是，该示例并没有整合类似 Eureka 之类的服务治理框架，所以需要将该参数设置为 false，否则配置的 serviceId 获取不到对应实例的清单。

- user-service.ribbon.listOfServers：该参数内容与 zuul.routes.<route>.serviceId 的配置相对应，开头的 user-service 对应了 serviceId 的值，这两个参数的配置相当于在该应用内部手工维护了服务与实例的对应关系。

不论是单实例还是多实例的配置方式，我们都需要为每一对映射关系指定一个名称，也就是上面配置中的<route>，每一个<route>对应了一条路由规则。每条路由规则都需要通过 path 属性来定义一个用来匹配客户端请求的路径表达式，并通过 url 或 serviceId 属性来指定请求表达式映射具体实例地址或服务名。

服务路由配置

对于服务路由，我们在快速入门示例中已经有过基础的介绍和体验，Spring Cloud Zuul 通过与 Spring Cloud Eureka 的整合，实现了对服务实例的自动化维护，所以在使用服务路由配置的时候，我们不需要向传统路由配置方式那样为 serviceId 指定具体的服务实例地址，只需要通过 zuul.routes.<route>.path 与 zuul.routes.<route>.serviceId 参数对的方式进行配置即可。

比如下面的示例，它实现了对符合/user-service/**规则的请求路径转发到名为 user-service 的服务实例上去的路由规则。其中<route>可以指定为任意的路由名称。

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service
```

对于面向服务的路由配置，除了使用 path 与 serviceId 映射的配置方式之外，还有一种更简洁的配置方式：zuul.routes.<serviceId>=<path>，其中<serviceId>用来指定路由的具体服务名，<path>用来配置匹配的请求表达式。比如下面的例子，它的路由规则等价于上面通过 path 与 serviceId 组合使用的配置方式。

```
zuul.routes.user-service=/user-service/**
```

传统路由的映射方式比较直观且容易理解，API 网关直接根据请求的 URL 路径找到最匹配的 path 表达式，直接转发给该表达式对应的 url 或对应 serviceId 下配置的实例地址，以实现外部请求的路由。那么当采用 path 与 serviceId 以服务路由的方式实现时，在没有配置任何实例地址的情况下，外部请求经过 API 网关的时候，它是如何被解析并转发到服务具体实例的呢？

在本章一开始，我们就提到了 Zuul 巧妙地整合了 Eureka 来实现面向服务的路由。实际上，我们可以直接将 API 网关也看作 Eureka 服务治理下的一个普通微服务应用。它除了会将自己注册到 Eureka 服务注册中心上之外，也会从注册中心获取所有服务以及它们的实例清单。所以，在 Eureka 的帮助下，API 网关服务本身就已经维护了系统中所有 serviceId

与实例地址的映射关系。当有外部请求到达 API 网关的时候，根据请求的 URL 路径找到最佳匹配的 path 规则，API 网关就可以知道要将该请求路由到哪个具体的 serviceId 上去。由于在 API 网关中已经知道 serviceId 对应服务实例的地址清单，那么只需要通过 Ribbon 的负载均衡策略，直接在这些清单中选择一个具体的实例进行转发就能完成路由工作了。

服务路由的默认规则

虽然通过 Eureka 与 Zuul 的整合已经为我们省去了维护服务实例清单的大量配置工作，剩下只需要再维护请求路径的匹配表达式与服务名的映射关系即可。但是在实际的运用过程中会发现，大部分的路由配置规则几乎都会采用服务名作为外部请求的前缀，比如下面的例子，其中 path 路径的前缀使用了 user-service，而对应的服务名称也是 user-service。

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service
```

对于这样具有规则性的配置内容，我们总是希望可以自动化地完成。非常庆幸，Zuul 默认实现了这样的贴心功能，当我们为 Spring Cloud Zuul 构建的 API 网关服务引入 Spring Cloud Eureka 之后，它为 Eureka 中的每个服务都自动创建一个默认路由规则，这些默认规则的 path 会使用 serviceId 配置的服务名作为请求前缀，就如上面的例子那样。

由于默认情况下所有 Eureka 上的服务都会被 Zuul 自动地创建映射关系来进行路由，这会使得一些我们不希望对外开放的服务也可能被外部访问到。这个时候，我们可以使用 zuul.ignored-services 参数来设置一个服务名匹配表达式来定义不自动创建路由的规则。Zuul 在自动创建服务路由的时候会根据该表达式来进行判断，如果服务名匹配表达式，那么 Zuul 将跳过该服务，不为其创建路由规则。比如，设置为 zuul.ignored-services=*的时候，Zuul 将对所有的服务都不自动创建路由规则。在这种情况下，我们就要在配置文件中逐个为需要路由的服务添加映射规则（可以使用 path 与 serviceId 组合的配置方式，也可使用更简洁的 zuul.routes.<serviceId>=<path>配置方式），只有在配置文件中出现的映射规则会被创建路由，而从 Eureka 中获取的其他服务，Zuul 将不会再为它们创建路由规则。

自定义路由映射规则

我们在构建微服务系统进行业务逻辑开发的时候，为了兼容外部不同版本的客户端程序（尽量不强迫用户升级客户端），一般都会采用开闭原则来进行设计与开发。这使得系统在迭代过程中，有时候会需要我们为一组互相配合的微服务定义一个版本标识来方便管理

它们的版本关系，根据这个标识我们可以很容易地知道这些服务需要一起启动并配合使用。比如可以采用类似这样的命名：userservice-v1、userservice-v2、orderservice-v1、orderservice-v2。默认情况下，Zuul 自动为服务创建的路由表达式会采用服务名作为前缀，比如针对上面的 userservice-v1 和 userservice-v2，它会产生/userservice-v1 和/userservice-v2 两个路径表达式来映射，但是这样生成出来的表达式规则较为单一，不利于通过路径规则来进行管理。通常的做法是为这些不同版本的微服务应用生成以版本代号作为路由前缀定义的路由规则，比如/v1/userservice/。这时候，通过这样具有版本号前缀的 URL 路径，我们就可以很容易地通过路径表达式来归类和管理这些具有版本信息的微服务了。

针对上面所述的需求，如果我们的各个微服务应用都遵循了类似 userservice-v1 这样的命名规则，通过-分隔的规范来定义服务名和服务版本标识的话，那么，我们可以使用 Zuul 中自定义服务与路由映射关系的功能，来实现为符合上述规则的微服务自动化地创建类似/v1/userservice/**的路由匹配规则。实现步骤非常简单，只需在 API 网关程序中，增加如下 Bean 的创建即可：

```
@Bean
public PatternServiceRouteMapper serviceRouteMapper() {
    return new PatternServiceRouteMapper(
            "(?<name>^.+)-(?<version>v.+$)",
            "${version}/${name}");
}
```

PatternServiceRouteMapper 对象可以让开发者通过正则表达式来自定义服务与路由映射的生成关系。其中构造函数的第一个参数是用来匹配服务名称是否符合该自定义规则的正则表达式，而第二个参数则是定义根据服务名中定义的内容转换出的路径表达式规则。当开发者在 API 网关中定义了 PatternServiceRouteMapper 实现之后，只要符合第一个参数定义规则的服务名，都会优先使用该实现构建出的路径表达式，如果没有匹配上的服务则还是会使用默认的路由映射规则，即采用完整服务名作为前缀的路径表达式。

路径匹配

不论是使用传统路由的配置方式还是服务路由的配置方式，我们都需要为每个路由规则定义匹配表达式，也就是上面所说的 path 参数。在 Zuul 中，路由匹配的路径表达式采用了 Ant 风格定义。

Ant 风格的路径表达式使用起来非常简单，它一共有下面这三种通配符。

通配符	说明
?	匹配任意单个字符
*	匹配任意数量的字符
**	匹配任意数量的字符，支持多级目录

我们可以通过下表中的示例来进一步理解这三个通配符的含义并进行参考使用。

URL 路径	说明
/user-service/?	它可以匹配/user-service/之后拼接一个任务字符的路径，比如/user-service/a、/user-service/b、/user-service/c
/user-service/*	它可以匹配/user-service/之后拼接任意字符的路径，比如/user-service/a、/user-service/aaa、/user-service/bbb。但是它无法匹配/user-service/a/b
/user-service/**	它可以匹配/user-service/*包含的内容之外，还可以匹配形如/user-service/a/b的多级目录路径

另外，当我们使用通配符的时候，经常会碰到这样的问题：一个 URL 路径可能会被多个不同路由的表达式匹配上。比如，有这样一个场景，我们在系统建设的一开始实现了 user-service 服务，并且配置了如下路由规则：

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service
```

但是随着版本的迭代，我们对 user-service 服务做了一些功能拆分，将原属于 user-service 服务的某些功能拆分到了另外一个全新的服务 user-service-ext 中去，而这些拆分的外部调用 URL 路径希望能够符合规则/user-service/ext/**，这个时候我们需要就在配置文件中增加一个路由规则，完整配置如下：

```
zuul.routes.user-service.path=/user-service/**
zuul.routes.user-service.serviceId=user-service

zuul.routes.user-service-ext.path=/user-service/ext/**
zuul.routes.user-service-ext.serviceId=user-service-ext
```

此时，调用 user-service-ext 服务的 URL 路径实际上会同时被/user-service/**和/user-service/ext/**两个表达式所匹配。在逻辑上，API 网关服务需要优先选择/user-service/ext/**路由，然后再匹配/user-service/**路由才能实现上述需求。但是如果使用上面的配置方式，实际上是无法保证这样的路由优先顺序的。

从下面的路由匹配算法中，我们可以看到它在使用路由规则匹配请求路径的时候是通过线性遍历的方式，在请求路径获取到第一个匹配的路由规则之后就返回并结束匹配过程。所以当存在多个匹配的路由规则时，匹配结果完全取决于路由规则的保存顺序。

```
    @Override
    public Route getMatchingRoute(final String path) {
        ...
        ZuulRoute route = null;
        if (!matchesIgnoredPatterns(adjustedPath)) {
            for (Entry<String, ZuulRoute> entry : this.routes.get().entrySet()) {
                String pattern = entry.getKey();
                log.debug("Matching pattern:" + pattern);
                if (this.pathMatcher.match(pattern, adjustedPath)) {
                    route = entry.getValue();
                    break;
                }
            }
        }
        log.debug("route matched=" + route);
        return getRoute(route, adjustedPath);
    }
```

下面所示的代码是基础的路由规则加载算法，我们可以看到这些路由规则是通过LinkedHashMap保存的，也就是说，路由规则的保存是有序的，而内容的加载是通过遍历配置文件中路由规则依次加入的，所以导致问题的根本原因是对配置文件中内容的读取。

```
    protected Map<String, ZuulRoute> locateRoutes() {
        LinkedHashMap<String, ZuulRoute> routesMap = new LinkedHashMap<String,
ZuulRoute>();
        for (ZuulRoute route : this.properties.getRoutes().values()) {
            routesMap.put(route.getPath(), route);
        }
        return routesMap;
    }
```

由于properties的配置内容无法保证有序，所以当出现这种情况的时候，为了保证路由的优先顺序，我们需要使用YAML文件来配置，以实现有序的路由规则，比如使用下面的定义：

```
zuul:
  routes:
    user-service-ext:
      path: /user-service/ext/**
      serviceId: user-service-ext
    user-service:
      path: /user-service/**
      serviceId: user-service
```

忽略表达式

通过 path 参数定义的 Ant 表达式已经能够完成 API 网关上的路由规则配置功能，但是为了更细粒度和更为灵活地配置路由规则，Zuul 还提供了一个忽略表达式参数 zuul.ignored-patterns。该参数可以用来设置不希望被 API 网关进行路由的 URL 表达式。

比如，以快速入门中的示例为基础，如果不希望/hello 接口被路由，那么我们可以这样设置：

```
zuul.ignored-patterns=/**/hello/**
zuul.routes.api-a.path=/api-a/**
zuul.routes.api-a.serviceId=hello-service
```

然后，可以尝试通过网关来访问 hello-service 的/hello 接口 http://localhost:5555/api-a/hello。虽然该访问路径完全符合 path 参数定义的 /api-a/**规则，但是由于该路径符合 zuul.ignored-patterns 参数定义的规则，所以不会被正确路由。同时，我们在控制台或日志中还能看到没有匹配路由的输出信息：

```
o.s.c.n.z.f.pre.PreDecorationFilter     : No route found for uri: /api-a/hello
```

另外，该参数在使用时还需要注意它的范围并不是对某个路由，而是对所有路由。所以在设置的时候需要全面考虑 URL 规则，防止忽略了不该被忽略的 URL 路径。

路由前缀

为了方便全局地为路由规则增加前缀信息，Zuul 提供了 zuul.prefix 参数来进行设置。比如，希望为网关上的路由规则都增加/api 前缀，那么我们可以在配置文件中增加配置：zuul.prefix=/api。另外，对于代理前缀会默认从路径中移除，我们可以通过设置 zuul.stripPrefix=false 来关闭该移除代理前缀的动作，也可以通过 zuul.routes.<route>.strip-prefix=true 来对指定路由关闭移除代理前缀的动作。

注意，在使用 zuul.prefix 参数的时候，目前版本的实现还存在一些 Bug，所以请谨慎使用，或是避开会引发 Bug 的配置规则。具体会引发 Bug 的规则如下：

假设我们设置 zuul.prefix=/api，当路由规则的 path 表达式以/api 开头的时候，将会产生错误的映射关系。可以进行下面的配置实验来验证这个问题：

```
zuul.routes.api-a.path=/api/a/**
zuul.routes.api-a.serviceId=hello-service

zuul.routes.api-b.path=/api-b/**
zuul.routes.api-b.serviceId=hello-service
```

```
zuul.routes.api-c.path=/ccc/**
zuul.routes.api-c.serviceId=hello-service
```

这里配置了三个路由关系：/api/a/**、/api-b/**、/ccc/**，这三个路径规则都将被路由到 hello-service 服务上去。当我们没有设置 zuul.prefix=/api 的时候，一切运作正常。但是在增加了 zuul.prefix=/api 配置之后，会得到下面这样的路由关系：

```
    o.s.c.n.zuul.web.ZuulHandlerMapping     : Mapped URL path [/api/api/a/a/**] onto
handler of type [class org.springframework.cloud.netflix.zuul.web.ZuulController]
    o.s.c.n.zuul.web.ZuulHandlerMapping     : Mapped URL path [/api/api-b-b/**] onto
handler of type [class org.springframework.cloud.netflix.zuul.web.ZuulController]
    o.s.c.n.zuul.web.ZuulHandlerMapping     : Mapped URL path [/api/ccc/**] onto
handler of type [class org.springframework.cloud.netflix.zuul.web.ZuulController]
```

从日志信息中我们可以看到，以/api 开头的路由规则解析除了两个看似就有问题的映射 URL，我们可以通过该输出的 URL 来访问，实际是路由不到正确的服务接口的，只有非/api 开头的路由规则/ccc/**能够正常路由。

上述实验基于 Brixton.SR7 和 Camden.SR3 测试均存在问题，所以在使用该版本或以下版本时候，务必避免让路由表达式的起始字符串与 zuul.prefix 参数相同。

本地跳转

在 Zuul 实现的 API 网关路由功能中，还支持 forward 形式的服务端跳转配置。实现方式非常简单，只需通过使用 path 与 url 的配置方式就能完成，通过 url 中使用 forward 来指定需要跳转的服务器资源路径。

下面的配置实现了两个路由规则，api-a 路由实现了将符合/api-a/**规则的请求转发到 http://localhost:8001/；而 api-b 路由则使用了本地跳转，它实现了将符合/api-b/**规则的请求转发到 API 网关中以/local 为前缀的请求上，由 API 网关进行本地处理。比如，当 API 网关接收到请求/api-b/hello，它符合 api-b 的路由规则，所以该请求会被 API 网关转发到网关的/local/hello 请求上进行本地处理。

```
zuul.routes.api-a.path=/api-a/**
zuul.routes.api-a.url=http://localhost:8001/

zuul.routes.api-b.path=/api-b/**
zuul.routes.api-b.url=forward:/local
```

这里要注意，由于需要在 API 网关上实现本地跳转，所以相应的我们也需要为本地跳转实现对应的请求接口。按照上面的例子，在 API 网关上还需要增加一个/local/hello 的接口实现才能让 api-b 路由规则生效，比如下面的实现。否则 Zuul 在进行 forward 转发的时候会因为找不到该请求而返回 404 错误。

```
@RestController
public class HelloController {

    @RequestMapping("/local/hello")
    public String hello() {
        return "Hello World Local";
    }

}
```

Cookie 与头信息

默认情况下，Spring Cloud Zuul 在请求路由时，会过滤掉 HTTP 请求头信息中的一些敏感信息，防止它们被传递到下游的外部服务器。默认的敏感头信息通过 `zuul.sensitiveHeaders` 参数定义，包括 `Cookie`、`Set-Cookie`、`Authorization` 三个属性。所以，我们在开发 Web 项目时常用的 `Cookie` 在 Spring Cloud Zuul 网关中默认是不会传递的，这就会引发一个常见的问题：如果我们要将使用了 Spring Security、Shiro 等安全框架构建的 Web 应用通过 Spring Cloud Zuul 构建的网关来进行路由时，由于 `Cookie` 信息无法传递，我们的 Web 应用将无法实现登录和鉴权。为了解决这个问题，配置的方法有很多。

- 通过设置全局参数为空来覆盖默认值，具体如下：

```
zuul.sensitiveHeaders=
```

  这种方法并不推荐，虽然可以实现 Cookie 的传递，但是破坏了默认设置的用意。在微服务架构的 API 网关之内，对于无状态的 RESTful API 请求肯定是要远多于这些 Web 类应用请求的，甚至还有一些架构设计会将 Web 类应用和 App 客户端一样都归为 API 网关之外的客户端应用。

- 通过指定路由的参数来配置，方法有下面两种。

```
# 方法一：对指定路由开启自定义敏感头
zuul.routes.<router>.customSensitiveHeaders=true
# 方法二：将指定路由的敏感头设置为空
zuul.routes.<router>.sensitiveHeaders=
```

  比较推荐使用这两种方法，仅对指定的 Web 应用开启对敏感信息的传递，影响范围小，不至于引起其他服务的信息泄露问题。

重定向问题

在解决了 Cookie 问题之后，我们已经能够通过网关来访问并登录到我们的 Web 应用

了。但是这个时候又会发现另外一个问题：虽然可以通过网关访问登录页面并发起登录请求，但是登录成功之后，我们跳转到的页面 URL 却是具体 Web 应用实例的地址，而不是通过网关的路由地址。这个问题非常严重，因为使用 API 网关的一个重要原因就是要将网关作为统一入口，从而不暴露所有的内部服务细节。那么是什么原因导致了这个问题呢？

通过浏览器开发工具查看登录以及登录之后的请求详情，可以发现，引起问题的大致原因是由于 Spring Security 或 Shiro 在登录完成之后，通过重定向的方式跳转到登录后的页面，此时登录后的请求结果状态码为 302，请求响应头信息中的 Location 指向了具体的服务实例地址，而请求头信息中的 Host 也指向了具体的服务实例 IP 地址和端口。所以，该问题的根本原因在于 Spring Cloud Zuul 在路由请求时，并没有将最初的 Host 信息设置正确。那么如何解决这个问题呢？

针对这个问题，目前在 `spring-cloud-netflix-core-1.2.x` 版本的 Zuul 中增加了一个参数配置，能够使得网关在进行路由转发前为请求设置 Host 头信息，以标识最初的服务端请求地址。具体配置方式如下：

```
zuul.addHostHeader=true
```

由于 Spring Cloud 的版本依赖原因，目前的 Brixton 版本采用了 `spring-cloud-netflix-core-1.1.x`，只有 Camden 版本采用了 `spring-cloud-netflix-core-1.2.x`。所以重定向的问题如果要在 Zuul 中解决，最简单的方法就使用 Camden 版本。如果要在 Brixton 版本中解决，可以参考 Camden 的 `PreDecorationFilter` 的实现扩展过滤器链来增加 Host 信息。

Hystrix 和 Ribbon 支持

在“快速入门”一节中介绍 `spring-cloud-starter-zuul` 依赖时，我们提到了它自身就包含了对 `spring-cloud-starter-hystrix` 和 `spring-cloud-starter-ribbon` 模块的依赖，所以 Zuul 天生就拥有线程隔离和断路器的自我保护功能，以及对服务调用的客户端负载均衡功能。但是需要注意，当使用 `path` 与 `url` 的映射关系来配置路由规则的时候，对于路由转发的请求不会采用 `HystrixCommand` 来包装，所以这类路由请求没有线程隔离和断路器的保护，并且也不会有负载均衡的能力。因此，我们在使用 Zuul 的时候尽量使用 `path` 和 `serviceId` 的组合来进行配置，这样不仅可以保证 API 网关的健壮和稳定，也能用到 Ribbon 的客户端负载均衡功能。

我们在使用 Zuul 搭建 API 网关的时候，可以通过 Hystrix 和 Ribbon 的参数来调整路由请求的各种超时时间等配置，比如下面这些参数的设置。

- `hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds`：该参数可以用来设置 API 网关中路由转发请求的 `HystrixCommand` 执行超时时间，单位为毫秒。当路由转发请求的命令执行时间超过该配置值之后，Hystrix 会将该执行命令标记为 `TIMEOUT` 并抛出异常，Zuul 会对该异常进行处理并返回如下 JSON 信息给外部调用方。

```
{
  "timestamp": 1481350975323,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "com.netflix.zuul.exception.ZuulException",
  "message": "TIMEOUT"
}
```

- `ribbon.ConnectTimeout`：该参数用来设置路由转发请求的时候，创建请求连接的超时时间。当 `ribbon.ConnectTimeout` 的配置值小于 `hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds` 配置值的时候，若出现路由请求出现连接超时，会自动进行重试路由请求，如果重试依然失败，Zuul 会返回如下 JSON 信息给外部调用方。

```
{
  "timestamp": 1481352582852,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "com.netflix.zuul.exception.ZuulException",
  "message": "NUMBEROF_RETRIES_NEXTSERVER_EXCEEDED"
}
```

  如果 `ribbon.ConnectTimeout` 的配置值大于 `hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds` 配置值的时候，当出现路由请求连接超时时，由于此时对于路由转发的请求命令已经超时，所以不会进行重试路由请求，而是直接按请求命令超时处理，返回 `TIMEOUT` 的错误信息。

- `ribbon.ReadTimeout`：该参数用来设置路由转发请求的超时时间。它的处理与 `ribbon.ConnectTimeout` 类似，只是它的超时是对请求连接建立之后的处理时间。当 `ribbon.ReadTimeout` 的配置值小于 `hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds` 配置值的时候，若路由请求的处理时间超过该配置值且依赖服务的请求还未响应的时候，会自动进

行重试路由请求。如果重试后依然没有获得请求响应，Zuul 会返回 NUMBEROF_RETRIES_NEXTSERVER_EXCEEDED 错误。如果 ribbon.ReadTimeout 的配置值大于 hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds 配置值，若路由请求的处理时间超过该配置值且依赖服务的请求还未响应时，不会进行重试路由请求，而是直接按请求命令超时处理，返回 TIMEOUT 的错误信息。

根据上面的介绍我们可以知道，在使用 Zuul 的服务路由时，如果路由转发请求发生超时（连接超时或处理超时），只要超时时间的设置小于 Hystrix 的命令超时时间，那么它就会自动发起重试。但是在有些情况下，我们可能需要关闭该重试机制，那么可以通过下面的两个参数来进行设置：

```
zuul.retryable=false
zuul.routes.<route>.retryable=false
```

其中，zuul.retryable 用来全局关闭重试机制，而 zuul.routes.<route>.retryable=false 则是指定路由关闭重试机制。

过滤器详解

在本章一开始的快速入门示例中，我们已经介绍了一部分关于请求过滤的功能。在本节中，我们将对 Zuul 的请求过滤器功能做进一步的介绍和总结。

过滤器

通过快速入门的示例，我们对于 Zuul 的第一印象通常是这样的：它包含了对请求的路由和过滤两个功能，其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部访问统一入口的基础；而过滤器功能则负责对请求的处理过程进行干预，是实现请求校验、服务聚合等功能的基础。然而实际上，路由功能在真正运行时，它的路由映射和请求转发都是由几个不同的过滤器完成的。其中，路由映射主要通过 pre 类型的过滤器完成，它将请求路径与配置的路由规则进行匹配，以找到需要转发的目标地址；而请求转发的部分则是由 route 类型的过滤器来完成，对 pre 类型过滤器获得的路由地址进行转发。所以，过滤器可以说是 Zuul 实现 API 网关功能最为核心的部件，每一个进入 Zuul 的 HTTP 请求都会经过一系列的过滤器处理链得到请求响应并返回给客户端。

在 Spring Cloud Zuul 中实现的过滤器必须包含 4 个基本特征：过滤类型、执行顺序、执行条件、具体操作。这些元素看起来非常熟悉，实际上它就是 ZuulFilter 接口中定义

的 4 个抽象方法：

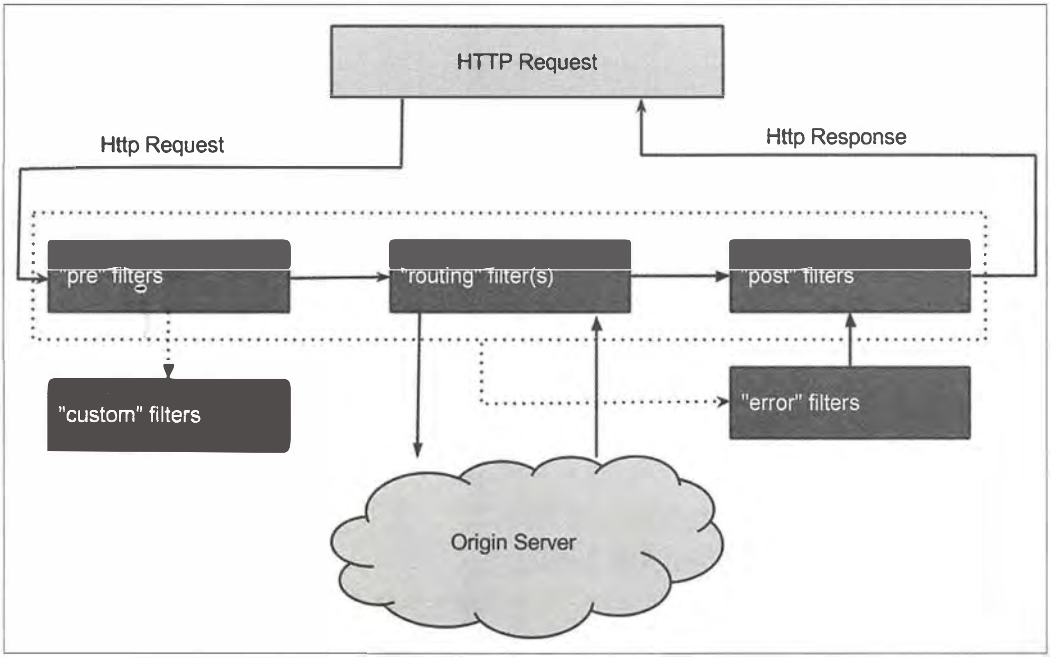
```
String filterType();

int filterOrder();

boolean shouldFilter();

Object run();
```

它们各自的含义与功能总结如下。

- filterType：该函数需要返回一个字符串来代表过滤器的类型，而这个类型就是在 HTTP 请求过程中定义的各个阶段。在 Zuul 中默认定义了 4 种不同生命周期的过滤器类型，具体如下所示。
    - pre：可以在请求被路由之前调用。
    - routing：在路由请求时被调用。
    - post：在 routing 和 error 过滤器之后被调用。
    - error：处理请求时发生错误时被调用。
- filterOrder：通过 int 值来定义过滤器的执行顺序，数值越小优先级越高。
- shouldFilter：返回一个 boolean 值来判断该过滤器是否要执行。我们可以通过此方法来指定过滤器的有效范围。
- run：过滤器的具体逻辑。在该函数中，我们可以实现自定义的过滤逻辑，来确定是否要拦截当前的请求，不对其进行后续的路由，或是在请求路由返回结果之后，对处理结果做一些加工等。

请求生命周期

对于 Zuul 中的过滤器类型 filterType，我们已经做过一些简单的介绍。Zuul 默认定义了 4 种不同的过滤器类型，它们覆盖了一个外部 HTTP 请求到达 API 网关，直到返回请求结果的全部生命周期。下图源自 Zuul 的官方 WiKi 中关于请求生命周期的图解，它描述了一个 HTTP 请求到达 API 网关之后，如何在各种不同类型的过滤器之间流转的详细过程。



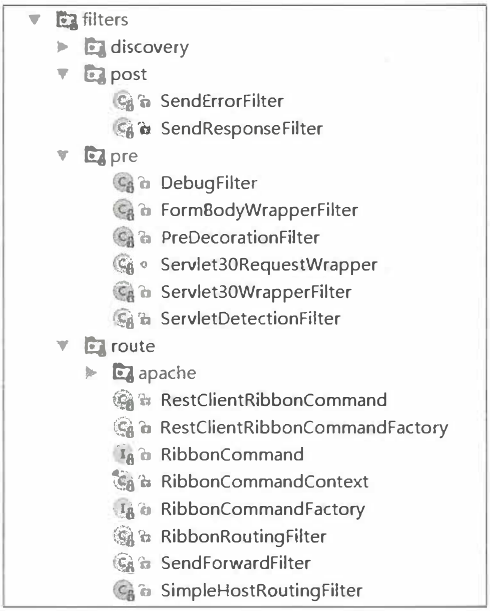
从上图中我们可以看到，当外部 HTTP 请求到达 API 网关服务的时候，首先它会进入第一个阶段 pre，在这里它会被 pre 类型的过滤器进行处理，该类型过滤器的主要目的是在进行请求路由之前做一些前置加工，比如请求的校验等。在完成了 pre 类型的过滤器处理之后，请求进入第二个阶段 routing，也就是之前说的路由请求转发阶段，请求将会被 routing 类型过滤器处理。这里的具体处理内容就是将外部请求转发到具体服务实例上去的过程，当服务实例将请求结果都返回之后，routing 阶段完成，请求进入第三个阶段 post。此时请求将会被 post 类型的过滤器处理，这些过滤器在处理的时候不仅可以获取到请求信息，还能获取到服务实例的返回信息，所以在 post 类型的过滤器中，我们可以对处理结果进行一些加工或转换等内容。另外，还有一个特殊的阶段 error，该阶段只有在上述三个阶段中发生异常的时候才会触发，但是它的最后流向还是 post 类型的过滤器，因为它需要通过 post 过滤器将最终结果返回给请求客户端（对于 error 过滤器的处理，在 Spring Cloud Zuul 的过滤链中实际上有一些不同，后续我们在介绍核心过滤器时会做详细分析）。

核心过滤器

在 Spring Cloud Zuul 中，为了让 API 网关组件可以被更方便地使用，它在 HTTP 请求生命周期的各个阶段默认实现了一批核心过滤器，它们会在 API 网关服务启动的时候被自动加载和启用。我们可以在源码中查看和了解它们，它们定义于 spring-cloud-

netflix-core 模块的 org.springframework.cloud.netflix.zuul.filters 包下。

如下图所示，在默认启用的过滤器中包含三种不同生命周期的过滤器，这些过滤器都非常重要，可以帮助我们理解 Zuul 对外部请求处理的过程，以及帮助我们在此基础上扩展过滤器去完成自身系统需要的功能。下面，我们将逐个对这些过滤器做详细的介绍。



pre 过滤器

- ServletDetectionFilter：它的执行顺序为-3，是最先被执行的过滤器。该过滤器总是会被执行，主要用来检测当前请求是通过 Spring 的 DispatcherServlet 处理运行的，还是通过 ZuulServlet 来处理运行的。它的检测结果会以布尔类型保存在当前请求上下文的 isDispatcherServletRequest 参数中，这样在后续的过滤器中，我们就可以通过 RequestUtils.isDispatcherServletRequest() 和 RequestUtils.isZuulServletRequest() 方法来判断请求处理的源头，以实现后续不同的处理机制。一般情况下，发送到 API 网关的外部请求都会被 Spring 的 DispatcherServlet 处理，除了通过 /zuul/* 路径访问的请求会绕过 DispatcherServlet，被 ZuulServlet 处理，主要用来应对处理大文件上传的情况。另外，对于 ZuulServlet 的访问路径 /zuul/*，我们可以通过 zuul.servletPath 参数来进行修改。

- Servlet30WrapperFilter：它的执行顺序为-2，是第二个执行的过滤器。目前的实现会对所有请求生效，主要为了将原始的 HttpServletRequest 包装成 Servlet30RequestWrapper 对象。
- FormBodyWrapperFilter：它的执行顺序为-1，是第三个执行的过滤器。该过滤器仅对两类请求生效，第一类是 Content-Type 为 application/x-www-form-urlencoded 的请求，第二类是 Content-Type 为 multipart/form-data 并且是由 Spring 的 DispatcherServlet 处理的请求（用到了 ServletDetectionFilter 的处理结果）。而该过滤器的主要目的是将符合要求的请求体包装成 FormBodyRequestWrapper 对象。
- DebugFilter：它的执行顺序为 1，是第四个执行的过滤器。该过滤器会根据配置参数 zuul.debug.request 和请求中的 debug 参数来决定是否执行过滤器中的操作。而它的具体操作内容则是将当前请求上下文中的 debugRouting 和 debugRequest 参数设置为 true。由于在同一个请求的不同生命周期中都可以访问到这两个值，所以我们在后续的各个过滤器中可以利用这两个值来定义一些 debug 信息，这样当线上环境出现问题的时候，可以通过请求参数的方式来激活这些 debug 信息以帮助分析问题。另外，对于请求参数中的 debug 参数，我们也可以通过 zuul.debug.parameter 来进行自定义。
- PreDecorationFilter：它的执行顺序为 5，是 pre 阶段最后被执行的过滤器。该过滤器会判断当前请求上下文中是否存在 forward.to 和 serviceId 参数，如果都不存在，那么它就会执行具体过滤器的操作（如果有一个存在的话，说明当前请求已经被处理过了，因为这两个信息就是根据当前请求的路由信息加载进来的）。而它的具体操作内容就是为当前请求做一些预处理，比如，进行路由规则的匹配、在请求上下文中设置该请求的基本信息以及将路由匹配结果等一些设置信息等，这些信息将是后续过滤器进行处理的重要依据，我们可以通过 RequestContext.getCurrentContext() 来访问这些信息。另外，我们还可以在该实现中找到一些对 HTTP 头请求进行处理的逻辑，其中包含了一些耳熟能详的头域，比如 X-Forwarded-Host、X-Forwarded-Port。另外，对于这些头域的记录是通过 zuul.addProxyHeaders 参数进行控制的，而这个参数的默认值为 true，所以 Zuul 在请求跳转时默认会为请求增加 X-Forwarded-*头域，包括 X-Forwarded-Host、X-Forwarded-Port、X-Forwarded-For、X-Forwarded-Prefix、X-Forwarded-Proto。也可以通过设置 zuul.addProxyHeaders=false 关闭对这些头域的添加动作。

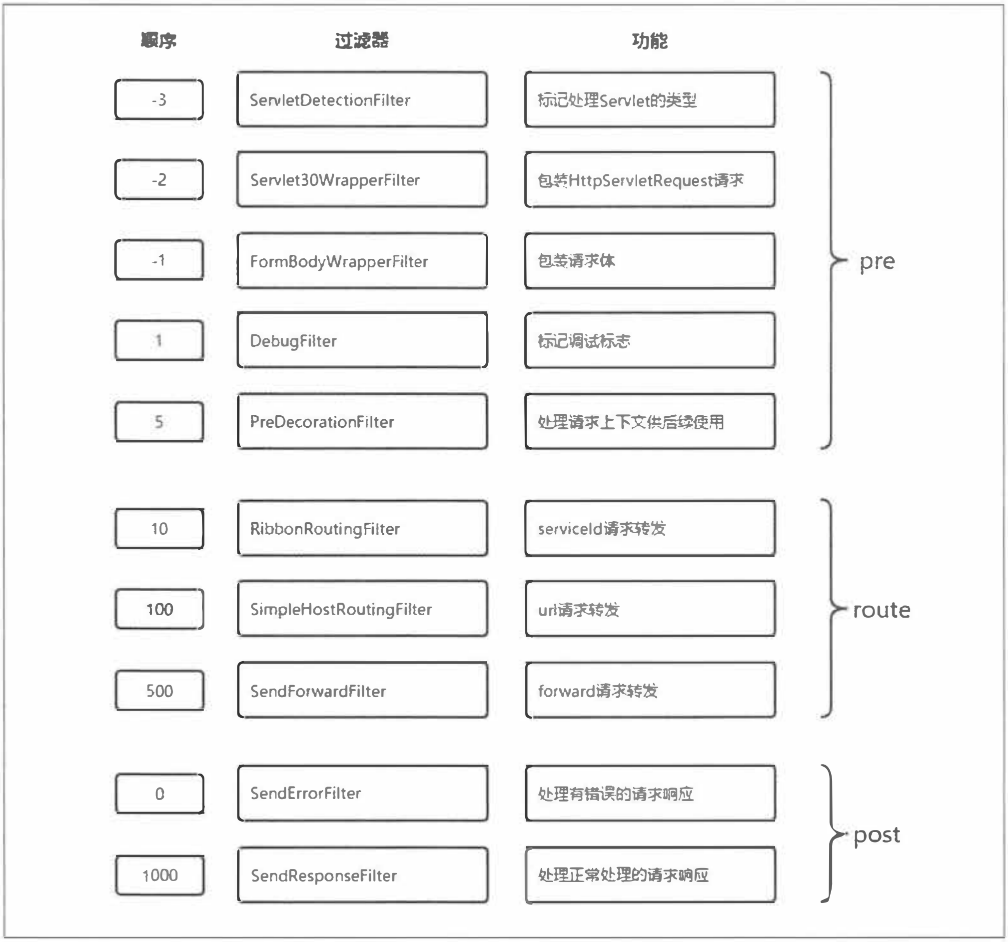
route 过滤器

- RibbonRoutingFilter：它的执行顺序为 10，是 route 阶段第一个执行的过滤器。该过滤器只对请求上下文中存在 serviceId 参数的请求进行处理，即只对通过 serviceId 配置路由规则的请求生效。而该过滤器的执行逻辑就是面向服务路由的核心，它通过使用 Ribbon 和 Hystrix 来向服务实例发起请求，并将服务实例的请求结果返回。
- SimpleHostRoutingFilter：它的执行顺序为 100，是 route 阶段第二个执行的过滤器。该过滤器只对请求上下文中存在 routeHost 参数的请求进行处理，即只对通过 url 配置路由规则的请求生效。而该过滤器的执行逻辑就是直接向 routeHost 参数的物理地址发起请求，从源码中我们可以知道该请求是直接通过 httpclient 包实现的，而没有使用 Hystrix 命令进行包装，所以这类请求并没有线程隔离和断路器的保护。
- SendForwardFilter：它的执行顺序为 500，是 route 阶段第三个执行的过滤器。该过滤器只对请求上下文中存在 forward.to 参数的请求进行处理，即用来处理路由规则中的 forward 本地跳转配置。

post 过滤器

- SendErrorFilter：它的执行顺序为 0，是 post 阶段第一个执行的过滤器。该过滤器仅在请求上下文中包含 error.status_code 参数（由之前执行的过滤器设置的错误编码）并且还没有被该过滤器处理过的时候执行。而该过滤器的具体逻辑就是利用请求上下文中的错误信息来组成一个 forward 到 API 网关/error 错误端点的请求来产生错误响应。
- SendResponseFilter：它的执行顺序为 1000，是 post 阶段最后执行的过滤器。该过滤器会检查请求上下文中是否包含请求响应相关的头信息、响应数据流或是响应体，只有在包含它们其中一个的时候执行处理逻辑。而该过滤器的处理逻辑就是利用请求上下文的响应信息来组织需要发送回客户端的响应内容。

下图对上述过滤器根据顺序、名称、功能、类型做了综合整理，可以帮助我们在自定义过滤器或是扩展过滤器的时候用来参考并全面地考虑整个请求生命周期的处理过程。



异常处理

通过上面请求生命周期和核心过滤器的介绍，我们会发现在核心过滤器中并没有实现 error 阶段的过滤器。那么当过滤器出现异常的时候需要如何处理呢？我们不妨在快速入门示例中做一个简单的试验，来看看过滤器中的异常需要被如何处理。

首先，我们尝试创建一个 pre 类型的过滤器，并在该过滤器的 run 方法实现中抛出一个异常。比如下面的实现，在 run 方法中调用的 doSomething 方法将抛出 RuntimeException 异常。

```
@Component
public class ThrowExceptionFilter extends ZuulFilter {

    private static Logger log =
```

```
LoggerFactory.getLogger(ThrowExceptionFilter.class);

        @Override
        public String filterType() {
            return "pre";
        }

        @Override
        public int filterOrder() {
            return 0;
        }

        @Override
        public boolean shouldFilter() {
            return true;
        }

        @Override
        public Object run() {
            log.info("This is a pre filter, it will throw a RuntimeException");
            doSomething();
            return null;
        }

        private void doSomething() {
            throw new RuntimeException("Exist some errors...");
        }

    }
```

注意在该类的定义上添加`@Component`注解，让 Spring 能够创建该过滤器实例，或者也可以像快速入门的例子中那样使用`@Bean`注解在应用主类中为其创建实例。

在添加了上面的过滤器之后，我们可以将该应用以及之前的相关应用运行起来，并根据网关配置的路由规则访问服务接口，比如`http://localhost:5555/api-a/hello`。此时我们会发现，在 API 网关服务的控制台中输出了`ThrowExceptionFilter`的过滤逻辑中的日志信息，但是并没有输出任何异常信息，同时发起的请求也没有获得任何响应结果。为什么会出现这样的情况呢？我们又该如何在过滤器中处理异常呢？翻遍 Spring Cloud Zuul 的文档以及 Netflix Zuul 的 Wiki 都没有找到相关的内容，但是这个问题却又是开发过程中普遍存在的。所以在本节中，我们将详细分析核心过滤器的异常处理机制以及如何在自定义过滤器中处理异常等内容。

try-catch 处理

回想一下，我们在上一节中介绍的所有核心过滤器，是否记得有一个 post 过滤器 SendErrorFilter 是用来处理异常信息的？根据正常的处理流程，该过滤器会处理异常信息，那么这里没有出现任何异常信息说明很有可能就是这个过滤器没有被执行。所以，不妨来详细看看 SendErrorFilter 的 shouldFilter 函数：

```
public boolean shouldFilter() {
    RequestContext ctx = RequestContext.getCurrentContext();
    return ctx.containsKey("error.status_code") && !ctx.getBoolean(SEND_ERROR_
FILTER_RAN, false);
}
```

可以看到，该方法的返回值中有一个重要的判断依据 ctx.containsKey("error.status_code")，也就是说请求上下文中必须有 error.status_code 参数，我们实现的 ThrowExceptionFilter 中并没有设置这个参数，所以自然不会进入 SendErrorFilter 过滤器的处理逻辑。那么如何使用这个参数呢？可以看一下 route 类型的几个过滤器，由于这些过滤器会对外发起请求，所以肯定会有异常需要处理，比如 RibbonRoutingFilter 的 run 方法实现如下：

```
public Object run() {
    RequestContext context = RequestContext.getCurrentContext();
    this.helper.addIgnoredHeaders();
    try {
        RibbonCommandContext commandContext = buildCommandContext(context);
        ClientHttpResponse response = forward(commandContext);
        setResponse(response);
        return response;
    }
    catch (ZuulException ex) {
        context.set(ERROR_STATUS_CODE, ex.nStatusCode);
        context.set("error.message", ex.errorCause);
        context.set("error.exception", ex);
    }
    catch (Exception ex) {
        context.set("error.status_code", HttpServletResponse.SC_INTERNAL_SERVER_
ERROR);
        context.set("error.exception", ex);
    }
    return null;
}
```

可以看到，整个发起请求的逻辑都采用了 try-catch 块处理。在 catch 异常的处理逻辑中并没有做任何输出操作，而是向请求上下文中添加了一些 error 相关的参数，主要有下面三个参数。

- error.status_code：错误编码。
- error.exception：Exception 异常对象。
- error.message：错误信息。

其中，error.status_code 参数就是 SendErrorFilter 过滤器用来判断是否需要执行的重要参数。分析到这里，实现异常处理的大致思路就开始明朗了，我们可以参考 RibbonRoutingFilter 的实现对 ThrowExceptionFilter 的 run 方法做一些异常处理的改造，具体如下：

```
public Object run() {
    log.info("This is a pre filter, it will throw a RuntimeException");
    RequestContext ctx = RequestContext.getCurrentContext();
    try {
        doSomething();
    } catch (Exception e) {
        ctx.set("error.status_code", HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        ctx.set("error.exception", e);
    }
    return null;
}
```

通过上面的改造之后，我们再尝试访问之前的接口，这个时候我们可以得到如下响应内容：

```
{
  "timestamp": 1481674980376,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "java.lang.RuntimeException",
  "message": "Exist some errors..."
}
```

此时，异常信息已经被 SendErrorFilter 过滤器正常处理并返回给客户端了，同时在网关的控制台中也输出了异常信息。从返回的响应信息中，可以看到几个之前我们在请求上下文中设置的内容，它们的对应关系如下所示。

- status：对应 error.status_code 参数的值。
- exception：对应 error.exception 参数中 Exception 的类型。
- message：对应 error.exception 参数中 Exception 的 message 信息。对于 message 的信息，我们在过滤器中还可以通过 ctx.set("error.message", "自定义异常消息");来定义更友好的错误信息。SendErrorFilter 会优先取 error.message 作为返回的 message 内容，如果没有的话才会使用 Exception 中的 message 信息。

ErrorFilter 处理

通过上面的分析与实验，我们已经知道如何在过滤器中正确处理异常，让错误信息能够顺利地流转到后续的 SendErrorFilter 过滤器来组织和输出。但是，即使我们不断强调要在过滤器中使用 try-catch 来处理业务逻辑并向请求上下文中添加异常信息，但是不可控的人为因素、意料之外的程序因素等，依然会使得一些异常从过滤器中抛出，对于意外抛出的异常又会导致没有控制台输出也没有任何响应信息的情况出现，那么是否有什么好的方法来为这些异常做一个统一的处理呢？

这个时候，我们就可以用到 error 类型的过滤器了。由于在请求生命周期的 pre、route、post 三个阶段中有异常抛出的时候都会进入 error 阶段的处理，所以可以通过创建一个 error 类型的过滤器来捕获这些异常信息，并根据这些异常信息在请求上下文中注入需要返回给客户端的错误描述。这里我们可以直接沿用在 try-catch 处理异常信息时用的那些 error 参数，这样就可以让这些信息被 SendErrorFilter 捕获并组织成响应消息返回给客户端。比如，下面的代码就实现了这里所描述的一个过滤器：

```
public class ErrorFilter extends ZuulFilter {

    Logger log = LoggerFactory.getLogger(ErrorFilter.class);

    @Override
    public String filterType() {
        return "error";
    }

    @Override
    public int filterOrder() {
        return 10;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        Throwable throwable = ctx.getThrowable();
        log.error("this is a ErrorFilter : {}", throwable.getCause().getMessage());
        ctx.set("error.status_code", HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
        ctx.set("error.exception", throwable.getCause());
```

```
        return null;
    }
}
```

在将该过滤器加入 API 网关服务之后，我们可以尝试使用之前介绍 `try-catch` 处理时实现的 `ThrowExceptionFilter`（不包含异常处理机制的代码），让该过滤器能够抛出异常。这个时候我们再通过 API 网关来访问服务接口。此时，我们就可以在控制台中看到 `ThrowExceptionFilter` 过滤器抛出的异常信息，并且请求响应中也能获得如下的错误信息内容，而不是什么信息都没有的情况了。

```
{
  "timestamp": 1481674993561,
  "status": 500,
  "error": "Internal Server Error",
  "exception": "java.lang.RuntimeException",
  "message": "Exist some errors..."
}
```

不足与优化

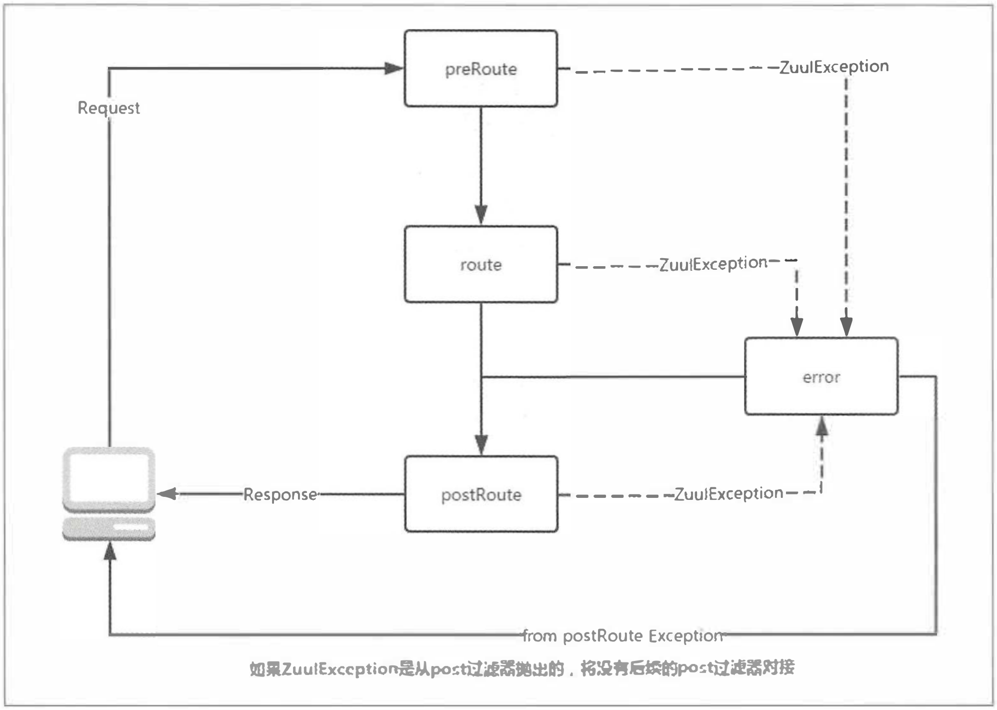
到这里，我们已经基本掌握了在核心过滤器处理逻辑之下，对自定义过滤器中处理异常的两种基本解决方法：一种是通过在各个阶段的过滤器中增加 `try-catch` 块，实现过滤器内部的异常处理；另一种是利用 `error` 类型过滤器的生命周期特性，集中处理 `pre`、`route`、`post` 阶段抛出的异常信息。通常情况下，我们可以将这两种手段同时使用，其中第一种是对开发人员的基本要求；而第二种是对第一种处理方式的补充，以防止意外情况的发生。

这样的异常处理机制看似已经完美，但是如果在多一些应用实践或源码分析之后，我们会发现依然存在一些不足。下面，我们不妨跟着源码来看看，到底上面的方案还有哪些不足之处需要注意和进一步优化。先来看看外部请求到达 API 网关服务之后，各个阶段的过滤器是如何进行调度的：

```
try {
   preRoute();
} catch (ZuulException e) {
   error(e);
   postRoute();
   return;
}
try {
   route();
} catch (ZuulException e) {
   error(e);
```

```
        postRoute();
        return;
    }
    try {
        postRoute();
    } catch (ZuulException e) {
        error(e);
        return;
    }
```

上面的代码源自 com.netflix.zuul.http.ZuulServlet 的 service 方法实现，它定义了 Zuul 处理外部请求过程时，各个类型过滤器的执行逻辑。从代码中我们可以看到三个 try-catch 块，它们依次分别代表了 pre、route、post 三个阶段的过滤器调用。在 catch 的异常处理中我们可以看到它们都会被 error 类型的过滤器进行处理（之前使用 error 过滤器来定义统一的异常处理也正是利用了这个特性）；error 类型的过滤器处理完毕之后，除了来自 post 阶段的异常之外，都会再被 post 过滤器进行处理。所以，上面代码中各个处理阶段的逻辑如下图所示：



通过图中的分析，我们可以看到，对于从 post 过滤器中抛出异常的情况，在经过 error 过滤器处理之后，就没有其他类型的过滤器来接手了，这就是使用之前所述方案存

在不足之处的根源。回想一下之前实现的两种异常处理方法，其中非常核心的一点是，这两种处理方法都在异常处理时向请求上下文中添加了一系列的 error.*参数，而这些参数真正起作用的地方是在 post 阶段的 SendErrorFilter，在该过滤器中会使用这些参数来组织内容返回给客户端。而对于 post 阶段抛出异常的情况下，由 error 过滤器处理之后并不会再调用 post 阶段的请求，自然这些 error.* 参数也就不会被 SendErrorFilter 消费输出。所以，如果我们在自定义 post 过滤器的时候，没有正确处理异常，就依然有可能出现日志中没有异常但请求响应内容为空的问题。我们可以通过将之前 ThrowExceptionFilter 的 filterType 修改为 post 来验证这个问题的存在，注意去掉 try-catch 块的处理，让它能够抛出异常。

解决上述问题的方法有很多种，最直接的是我们可以在实现 error 过滤器的时候，直接组织结果返回就能实现效果。但是这样做的缺点也很明显，对于错误信息组织和返回的代码实现会存在多份，这样非常不利于日后的代码维护工作。所以为了保持对异常返回处理逻辑的一致性，我们还是希望将 post 过滤器抛出的异常交给 SendErrorFilter 来处理。

在前文中，我们已经实现了一个 ErrorFilter 来捕获 pre、route、post 过滤器抛出的异常，并组织 error.*参数保存到请求的上下文中。由于我们的目标是沿用 SendErrorFilter，这些 error.*参数依然对我们有用，所以可以继续沿用该过滤器，让它在 post 过滤器抛出异常的时候，继续组织 error.*参数，只是这里我们已经无法将这些 error.*参数再传递给 SendErrorFilter 过滤器来处理了。所以，我们需要在 ErrorFilter 过滤器之后再定义一个 error 类型的过滤器，让它来实现 SendErrorFilter 的功能，但是这个 error 过滤器并不需要处理所有出现异常的情况，它仅对 post 过滤器抛出的异常有效。根据上面的思路，我们完全可以创建一个继承自 SendErrorFilter 的过滤器，复用它的 run 方法，然后重写它的类型、顺序以及执行条件，实现对原有逻辑的复用，具体实现如下：

```
@Component
public class ErrorExtFilter extends SendErrorFilter {

    @Override
    public String filterType() {
        return "error";
    }

    @Override
    public int filterOrder() {
        return 30;  // 大于ErrorFilter的值
    }
```

```
    @Override
    public boolean shouldFilter() {
        // TODO 判断：仅处理来自post过滤器引起的异常
        return true;
    }

}
```

到这里，我们在过滤器调度上的实现思路已经很清晰了，但是又有一个问题出现在我们面前：怎么判断引起异常的过滤器来自什么阶段呢？shouldFilter 方法该如何实现，对于这个问题，我们第一反应会寄希望于请求上下文 RequestContext 对象，可是在查阅文档和源码后发现其中并没有存储异常来源的内容，所以我们不得不扩展原来的过滤器处理逻辑。当有异常抛出的时候，记录下抛出异常的过滤器，这样我们就可以在 ErrorExtFilter 过滤器的 shouldFilter 方法中获取并以此判断异常是否来自 post 阶段的过滤器了。

为了扩展过滤器的处理逻辑，为请求上下文增加一些自定义属性，我们需要深入了解 Zuul 过滤器的核心处理器：com.netflix.zuul.FilterProcessor。该类中定义了下面列出的过滤器调用和处理相关的核心方法。

- getInstance()：该方法用来获取当前处理器的实例。
- setProcessor(FilterProcessor processor)：该方法用来设置处理器实例，可以使用此方法来设置自定义的处理器。
- processZuulFilter(ZuulFilter filter)：该方法定义了用来执行 filter 的具体逻辑，包括对请求上下文的设置，判断是否应该执行，执行时一些异常的处理等。
- getFiltersByType(String filterType)：该方法用来根据传入的 filterType 获取 API 网关中对应类型的过滤器，并根据这些过滤器的 filterOrder 从小到大排序，组织成一个列表返回。
- runFilters(String sType)：该方法会根据传入的 filterType 来调用 getFiltersByType(String filterType)获取排序后的过滤器列表，然后轮询这些过滤器，并调用 processZuulFilter(ZuulFilter filter)来依次执行它们。
- preRoute()：调用 runFilters("pre")来执行所有 pre 类型的过滤器。
- route()：调用 runFilters("route")来执行所有 route 类型的过滤器。
- postRoute()：调用 runFilters("post")来执行所有 post 类型的过滤器。
- error()：调用 runFilters("error")来执行所有 error 类型的过滤器。

根据之前的设计，可以直接扩展 processZuulFilter(ZuulFilter filter)，当过滤器执行抛出异常的时候，我们捕获它，并向请求上下中记录一些信息。比如下面的具体实现：

```
public class DidiFilterProcessor extends FilterProcessor {

    @Override
    public Object processZuulFilter(ZuulFilter filter) throws ZuulException {
        try {
            return super.processZuulFilter(filter);
        } catch (ZuulException e) {
            RequestContext ctx = RequestContext.getCurrentContext();
            ctx.set("failed.filter", filter);
            throw e;
        }
    }

}
```

在上面代码的实现中，我们创建了一个 FilterProcessor 的子类，并重写了 processZuulFilter(ZuulFilter filter)，虽然主逻辑依然使用了父类的实现，但是在最外层，我们为其增加了异常捕获，并在异常处理中为请求上下文添加了 failed.filter 属性，以存储抛出异常的过滤器实例。在实现了这个扩展之后，我们也就可以完善之前 ErrorExtFilter 中的 shouldFilter()方法了，通过从请求上下文中获取该信息做出正确的判断，具体实现如下：

```
@Component
public class ErrorExtFilter extends SendErrorFilter {

    @Override
    public String filterType() {
        return "error";
    }

    @Override
    public int filterOrder() {
        return 30;  // 大于ErrorFilter的值
    }

    @Override
    public boolean shouldFilter() {
        // 判断：仅处理来自post过滤器引起的异常
        RequestContext ctx = RequestContext.getCurrentContext();
        ZuulFilter failedFilter = (ZuulFilter) ctx.get("failed.filter");
```

```
        if(failedFilter != null && failedFilter.filterType().equals("post")) {
            return true;
        }
        return false;
    }

}
```

到这里，我们的优化任务还没有完成，因为扩展的过滤器处理类并还没有生效。最后，需要在应用主类中，通过调用 `FilterProcessor.setProcessor(new DidiFilter-Processor());`方法来启用自定义的核心处理器以完成我们的优化目标。

自定义异常信息

在实现了对自定义过滤器中的异常处理之后，实际应用到业务系统中时，往往默认的错误信息并不符合系统设计的响应格式，那么我们就需要对返回的异常信息进行定制。对于如何定制这个错误信息有很多种方法可以实现。最直接的是，可以编写一个自定义的 post 过滤器来组织错误结果，该方法实现起来简单粗暴，完全可以参考 SendErrorFilter 的实现，然后直接组织请求响应结果而不是 forward 到/error 端点，只是使用该方法时需要注意：为了替代 SendErrorFilter，还需要禁用 SendErrorFilter 过滤器，禁用的配置方法在后文中会详细介绍。

那么如果不采用重写过滤器的方式，依然想要使用 SendErrorFilter 来处理异常返回的话，我们要如何定制返回的响应结果呢？这个时候，我们的关注点就不能放在 Zuul 的过滤器上了，因为错误信息的生成实际上并不是由 Spring Cloud Zuul 完成的。我们在介绍 SendErrorFilter 的时候提到过，它会根据请求上下中保存的错误信息来组织一个 forward 到/error 端点的请求来获取错误响应，所以我们的扩展目标转移到了对/error 端点的实现。

/error 端点的实现来源于 Spring Boot 的 org.springframework.boot.autoconfigure.web.BasicErrorController，它的具体定义如下：

```
@RequestMapping
@ResponseBody
public ResponseEntity<Map<String, Object>> error(HttpServletRequest request) {
    Map<String, Object> body = getErrorAttributes(request,
            isIncludeStackTrace(request, MediaType.ALL));
    HttpStatus status = getStatus(request);
    return new ResponseEntity<Map<String, Object>>(body, status);
}
```

从源码中可以看到，它的实现非常简单，通过调用 getErrorAttributes 方法来根据请求参数组织错误信息的返回结果，而这里的 getErrorAttributes 方法会将具体组织逻

辑委托给 org.springframework.boot.autoconfigure.web.ErrorAttributes 接口提供的 getErrorAttributes 来实现。在 Spring Boot 的自动化配置机制中，默认会采用 org.springframework.boot.autoconfigure.web.DefaultErrorAttributes 作为该接口的实现。如下代码所示，在定义 Error 处理的自动化配置中，该接口的默认实现采用了@ConditionalOnMissingBean 修饰，说明 DefaultErrorAttributes 对象实例仅在没有 ErrorAttributes 接口的实例时才会被创建来使用，所以我们只需要自己编写一个自定义的 ErrorAttributes 接口实现类，并创建它的实例就能替代这个默认的实现，从而达到自定义错误信息的效果了。

```
    @ConditionalOnClass({ Servlet.class, DispatcherServlet.class })
    @ConditionalOnWebApplication
    @AutoConfigureBefore(WebMvcAutoConfiguration.class)
    @Configuration
    public class ErrorMvcAutoConfiguration {

      @Autowired
      private ServerProperties properties;

      @Bean
      @ConditionalOnMissingBean(value = ErrorAttributes.class, search =
SearchStrategy.CURRENT)
      public DefaultErrorAttributes errorAttributes() {
        return new DefaultErrorAttributes();
      }
      ...
    }
```

举个简单的例子，比如我们不希望将 exception 属性返回给客户端，那么就可以编写一个自定义的实现，它可以基于 DefaultErrorAttributes，然后重写 getErrorAttributes 方法，从原来的结果中将 exception 移除即可，具体实现如下：

```
    public class DidiErrorAttributes extends DefaultErrorAttributes {

       @Override
       public Map<String, Object> getErrorAttributes (
             RequestAttributes requestAttributes, boolean includeStackTrace){
          Map<String, Object> result = super.getErrorAttributes(requestAttributes,
includeStackTrace);
          result.remove("exception");
          return result;
       }

    }
```

最后，为了让自定义的错误信息生成逻辑生效，需要在应用主类中加入如下代码，为其创建实例来替代默认的实现：

```
@Bean
public DefaultErrorAttributes errorAttributes() {
    return new DidiErrorAttributes();
}
```

通过上面介绍的方法，我们就能基于Zuul的核心过滤器来灵活地自定义错误返回信息，以满足实际应用系统的响应格式了。

禁用过滤器

不论是核心过滤器还是自定义过滤器，只要在API网关应用中为它们创建了实例，那么默认情况下，它们都是启用状态的。那么如果有些过滤器我们不想使用了，如何禁用它们呢？大多情况下初识Zuul的使用者第一反应就是通过重写 shouldFilter 逻辑，让它返回 false，这样该过滤器对于任何请求都不会被执行，基本实现了对过滤器的禁用。但是，对于自定义过滤器来说似乎是实现了过滤器不生效的功能，但是这样的做法缺乏灵活性。由于直接要修改过滤器逻辑，我们不得不重新编译程序，并且如果该过滤器在未来一段时间还有可能被启用的时候，那么就又得修改代码并编译程序。同时，对于核心过滤器来说，就更为麻烦，我们不得不获取源码来进行修改和编译。

实际上，在Zuul中特别提供了一个参数来禁用指定的过滤器，该参数的配置格式如下：

```
zuul.<SimpleClassName>.<filterType>.disable=true
```

其中，<SimpleClassName>代表过滤器的类名，比如快速入门示例中的AccessFilter；<filterType>代表过滤器类型，比如快速入门示例中 AccessFilter 的过滤器类型 pre。所以，如果我们想要禁用快速入门示例中的 AccessFilter 过滤器，只需要在 application.properties 配置文件中增加如下配置即可：

```
zuul.AccessFilter.pre.disable=true
```

该参数配置除了可以对自定义的过滤器进行禁用配置之外，很多时候可以用它来禁用 Spring Cloud Zuul 中默认定义的核心过滤器。这样我们就可以抛开 Spring Cloud Zuul 自带的那套核心过滤器，实现一套更符合我们实际需求的处理机制。

通过本节对异常处理的介绍，也许这些方法并一定完全适用读者所接触的实际系统，但是通过这些分析可以帮助我们进一步理解 Zuul 过滤器的运行机制，帮助我们基于 Spring Cloud Zuul 去实现更适合自身系统的 API 网关服务。

动态加载

在微服务架构中，由于 API 网关服务担负着外部访问统一入口的重任，它同其他应用不同，任何关闭应用和重启应用的操作都会使系统对外服务停止，对于很多 7 × 24 小时服务的系统来说，这样的情况是绝对不被允许的。所以，作为最外部的网关，它必须具备动态更新内部逻辑的能力，比如动态修改路由规则、动态添加/删除过滤器等。

通过 Zuul 实现的 API 网关服务当然也具备了动态路由和动态过滤器的能力。我们可以在不重启 API 网关服务的前提下，为其动态修改路由规则和添加或删除过滤器。下面我们分别来看看如何通过 Zuul 来实现动态 API 网关服务。

动态路由

通过之前对请求路由的详细介绍，我们可以发现对于路由规则的控制几乎都可以在配置文件 `application.properties` 或 `application.yaml` 中完成。既然这样，对于如何实现 Zuul 的动态路由，我们很自然地会将它与 Spring Cloud Config 的动态刷新机制联系到一起。只需将 API 网关服务的配置文件通过 Spring Cloud Config 连接的 Git 仓库存储和管理，我们就能轻松实现动态刷新路由规则的功能。

在介绍如何具体实现 API 网关服务的动态路由之前，我们首先需要一个连接到 Git 仓库的分布式配置中心 `config-server` 应用。如果还没有搭建过分布式配置中心的话，建议先阅读第 8 章的内容，对分布式配置中心的运作机制有一个基础的了解，并构建一个 `config-server` 应用，以配合完成下面的内容。

在具备了分布式配置中心之后，为了方便理解，我们重新构建一个 API 网关服务，该服务的配置中心不再配置于本地工程中，而是从 `config-server` 中获取，构建过程如下所示。

- 创建一个基础的 Spring Boot 工程，命名为 `api-gateway-dynamic-route`。
- 在 `pom.xml` 中引入对 `zuul`、`eureka` 和 `config` 的依赖，具体内容如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zuul</artifactId>
```

```
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-eureka</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Brixton.SR5</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

- 在/resource 目录下创建配置文件 bootstrap.properties，并在该文件中指定 config-server 和 eureka-server 的具体地址，以获取应用的配置文件和实现服务注册与发现。

```
spring.application.name=api-gateway
server.port=5556

spring.cloud.config.uri=http://localhost:7001/

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

- 创建用来启动 API 网关的应用主类。这里我们需要使用@RefreshScope 注解来将 Zuul 的配置内容动态化，具体实现如下：

```
@EnableZuulProxy
@SpringCloudApplication
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

    @Bean
```

```
    @RefreshScope
    @ConfigurationProperties("zuul")
    public ZuulProperties zuulProperties() {
        return new ZuulProperties();
    }
}
```

- 在完成了所有程序相关的编写之后，我们还需要在 Git 仓库中增加网关的配置文件，取名为 api-gateway.properties。在配置文件中，我们为 API 网关服务预定义以下路由规则，比如：

```
zuul.routes.service-a.path=/service-a/**
zuul.routes.service-a.serviceId=hello-service
zuul.routes.service-b.path=/service-b/**
zuul.routes.service-b.url=http://localhost:8001/
```

  对于 API 网关服务在 Git 仓库中的配置文件名称完全取决于网关应用配置文件 bootstrap.properties 中 spring.application.name 属性的配置值。由于本章我们主要介绍 API 网关的使用，所以这里省略了关于 label 和 profile 的配置，默认会使用 master 分支和 default 配置文件，更细致的配置方式读者可查阅第 8 章介绍的内容。

测试与验证

在完成了上述内容之后，我们可以将 config-server、eureka-server、api-gateway-dynamic-route 以及配置文件中路由规则指向的具体服务，比如 hello-service 启动起来。此时，在 API 网关应用 api-gateway-dynamic-route 的控制台中，我们可以看到它输出了从 config-server 中获取配置文件过程的日志信息，根据这些信息，可以判断获取配置信息的路径等内容是否正确。具体输出内容如下所示：

```
    c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at: 
http://localhost:7001/
    c.c.c.ConfigServicePropertySourceLocator : Located environment: name=api-gateway, 
profiles=[default], label=master, version=1dab6126ca2972c5409fcb089934b057cf2bf77d
    b.c.PropertySourceBootstrapConfiguration : Located property source: 
CompositePropertySource [name='configService', propertySources=[MapPropertySource 
[name='overrides'], MapPropertySource 
[name='http://git.oschina.net/didispace/SpringCloud-Learning/spring_cloud_in_action
/config-repo/api-gateway.properties']]]
    com.didispace.Application               : No active profile set, falling back to 
default profiles: default
```

在 api-gateway-dynamic-route 启动完成之后，可以通过对 API 网关服务调用 /routes 接口来获取当前网关上的路由规则，根据上述配置我们可以得到如下返回信息：

```
{
  "/service-a/**": "hello-service",
  "/service-b/**": "http://localhost:8001/"
}
```

我们可以先尝试着通过上述路由规则访问一下对应路由服务提供的接口，验证当前的路由规则是否生效。接着，我们开始尝试动态修改这些路由规则，只需要以下两步。

- 修改 Git 仓库中的 `api-gateway.properties` 配置文件，比如，将 `service-a` 路由规则的 `path` 修改为`/aaa/**`，并把 `service-b` 的路由规则从 `url` 修改为 `serviceId` 方式。具体配置如下：

```
zuul.routes.service-a.path=/aaa/**
zuul.routes.service-a.serviceId=hello-service

zuul.routes.service-b.path=/service-b/**
zuul.routes.service-b.url=hello-service
```

- 在修改完配置文件之后，将修改内容推送到远程仓库。然后，通过向 `api-gateway-dynamic-route` 的`/refresh` 接口发送 POST 请求来刷新配置信息。当配置文件有修改的时候，该接口会返回被修改的属性名称，根据上面的修改，我们会得到如下返回信息：

```
[
   "zuul.routes.service-b.serviceId",
   "zuul.routes.service-b.url",
   "zuul.routes.service-a.path"
]
```

  由于修改了 `service-b` 路由的 `url` 方式为 `serviceId` 方式，相当于删除了 `url` 参数配置，增加了 `serviceId` 参数配置，所以这里会出现两条关于 `service-b` 路由的变更信息。

到这里，我们就已经完成了路由规则的动态刷新，可以继续通过 API 网关服务的 `/routes` 接口来查看当前的所有路由规则，该接口将返回如下信息：

```
{
  "/aaa/**": "hello-service",
  "/service-b/**": "hello-service"
}
```

从`/routes` 接口的信息中我们可以看到，路由 `service-a` 的匹配表达式被修改成了`/aaa/**`，而路由 `service-b` 的映射目标从原来的物理地址 `http://localhost:8001/` 修改成了 `hello-service` 服务。

通过本节对动态路由加载内容的介绍，我们可以看到，通过 Zuul 构建的 API 网关服务对于动态路由的实现总体上来说还是非常简单的。美中不足的一点是，Spring Cloud Config 并没有 UI 管理界面，我们不得不通过 Git 客户端来进行修改和配置，所以在使用的时候并不是特别方便，当然有条件的团队可以自己开发一套 UI 界面来帮助管理这些路由规则。

动态过滤器

既然通过 Zuul 构建的 API 网关服务能够轻松地实现动态路由的加载，那么对于 API 网关服务的另外一大重要功能——请求过滤器的动态加载自然也不能放过，只是对于请求过滤器的动态加载与请求路由的动态加载在实现机制上会有所不同。这个不难理解，通过之前介绍的请求路由和请求过滤的示例，我们可以看到请求路由通过配置文件就能实现，而请求过滤则都是通过编码实现。所以，对于实现请求过滤器的动态加载，我们需要借助基于 JVM 实现的动态语言的帮助，比如 Groovy。

下面，我们将通过一个简单的示例来演示如何构建一个具备动态加载 Groovy 过滤器能力的 API 网关服务的详细步骤。

- 创建一个基础的 Spring Boot 工程，命名为 `api-gateway-dynamic-filter`。
- 在 `pom.xml` 中引入对 `zuul`、`eureka` 和 `groovy` 的依赖，具体内容如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-zuul</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
    <dependency>
        <groupId>org.codehaus.groovy</groupId>
        <artifactId>groovy-all</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
```

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- 在/resource 目录下创建配置文件 application.properties，并在该文件中设置API网关服务的应用名和服务端口号，以及指定eureka-server的具体地址。同时，再配置一个用于测试的路由规则，我们可以用之前章节实现的 hello-service 为例。具体配置内容如下：

```
spring.application.name=api-gateway
server.port=5555

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/

zuul.routes.hello.path=/hello-service/**
zuul.routes.hello.serviceId=hello-service
```

到这里一个基础的 API 网关服务已经构建完成，下面我们来为它增加动态加载过滤器的功能。

- 为了方便使用，我们先自定义一些用来配置动态加载过滤器的参数，并将它们的配置值加入到 application.properties 中，比如：

```
zuul.filter.root=filter
zuul.filter.interval=5
```

  其中，zuul.filter.root 用来指定动态加载的过滤器存储路径；zuul.filter.interval 用来配置动态加载的间隔时间，以秒为单位。

- 创建用来加载自定义属性的配置类，命名为 FilterConfiguration，具体内容如下：

```
@ConfigurationProperties("zuul.filter")
public class FilterConfiguration {

  private String root;
  private Integer interval;

  public String getRoot() {
      return root;
```

```
    }

    public void setRoot(String root) {
        this.root = root;
    }

    public Integer getInterval() {
        return interval;
    }

    public void setInterval(Integer interval) {
        this.interval = interval;
    }
}
```

- 创建应用启动主类，并在该类中引入上面定义的 `FilterConfiguration` 配置，并创建动态加载过滤器的实例。具体内容如下：

```
@EnableZuulProxy
@EnableConfigurationProperties({FilterConfiguration.class})
@SpringCloudApplication
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

    @Bean
    public FilterLoader filterLoader(FilterConfiguration filterConfiguration) {
        FilterLoader filterLoader = FilterLoader.getInstance();
        filterLoader.setCompiler(new GroovyCompiler());
        try {
            FilterFileManager.setFilenameFilter(new GroovyFileFilter());
            FilterFileManager.init(
                    filterConfiguration.getInterval(),
                    filterConfiguration.getRoot() + "/pre",
                    filterConfiguration.getRoot() + "/post");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return filterLoader;
    }

}
```

至此，我们就已经完成了为基础的 API 网关服务增加动态加载过滤器的能力。根据上

面的定义，API 网关应用会每隔 5 秒，从 API 网关服务所在位置的 filter/pre 和 filter/post 目录下获取 Groovy 定义的过滤器，并对其进行编译和动态加载使用。对于动态加载的时间间隔，可通过 zuul.filter.interval 参数来修改。而加载过滤器实现类的根目录可通过 zuul.filter.root 调整根目录的位置来修改，但是对于根目录的子目录，这里写死了读取/pre 和/post 目录，实际使用的时候读者可以做进一步扩展。

在完成上述构建之后，我们可以将涉及的服务，比如 eureka-server、hello-service 以及上述实现的 API 网关服务都启动起来。在没有加入任何自定义过滤器的时候，根据路由规则定义，我们可以尝试向 API 网关服务发起请求：http://localhost:5555/hello-service/hello，如果配置正确，该请求会被 API 网关服务路由到 hello-service 上，并返回输出 Hello World。

接下来，我们可以在 filter/pre 和 filter/post 目录下，用 Groovy 来创建一些过滤器来看看实际效果，举例如下。

- 在 filter/pre 目录下创建一个 pre 类型的过滤器，命名为 PreFilter.groovy。由于 pre 类型的过滤器在请求路由前执行，通常用来做一些签名校验的功能，所以我们可以在过滤器中输出一些请求相关的信息，比如下面的实现：

```
class PreFilter extends ZuulFilter {
  Logger log = LoggerFactory.getLogger(PreFilter.class)

  @Override
  String filterType() {
    return "pre"
  }

  @Override
  int filterOrder() {
    return 1000
  }

  @Override
  boolean shouldFilter() {
    return true
  }

  @Override
  Object run() {
    HttpServletRequest request = RequestContext.getCurrentContext().getRequest()
    log.info("this is a pre filter: Send {} request to {}",
           request.getMethod(),
```

```
            request.getRequestURL().toString())
        return null
    }
}
```

在加入了该过滤器之后，不需要重启 API 网关服务，只需要稍等几秒就会生效。我们可以继续尝试向 API 网关服务发起请求：http://localhost:5555/hello-service/hello，此时在控制台中可以看到 PreFilter.groovy 过滤器中定义的日志信息，具体如下：

```
    com.didispace.filter.pre.PreFilter      : this is a pre filter: Send GET request to
http://localhost:5555/ddd/hello
```

- 在 filter/post 目录下创建一个 post 类型的过滤器，命名为 PostFilter.groovy。由于 post 类型的过滤器在请求路由返回后执行，我们可以进一步对这个结果做一些处理，对微服务返回的信息做一些加工。比如下面的实现：

```
class PostFilter extends ZuulFilter{
  Logger log = LoggerFactory.getLogger(PostFilter.class)

  @Override
  String filterType() {
    return "post"
  }

  @Override
  int filterOrder() {
    return 2000
  }

  @Override
  boolean shouldFilter() {
    return true
  }

  @Override
  Object run() {
    log.info("this is a post filter: Receive response")
    HttpServletResponse response = RequestContext.getCurrentContext().getResponse()
    response.getOutputStream().print(", I am zhaiyongchao")
    response.flushBuffer()
  }
}
```

在加入了该过滤器之后，我们也不需要重启 API 网关服务，稍等几秒后就可以尝试向 API 网关服务发起请求：http://localhost:5555/hello-service/hello，

此时不仅可以在控制台中看到 PostFilter.groovy 过滤器中定义的日志输出信息，也可以从请求返回的内容发现过滤器的效果，该接口返回的内容不再是 Hello World，而是经过加工处理后的 Hello World, I am zhaiyongchao。

通过本节对动态过滤器加载的内容的介绍，可以看到，API 网关服务的动态过滤器功能可以帮助我们增强 API 网关的持续服务能力，对于网关中的处理逻辑维护也变得更为灵活，不仅可以动态地实现请求校验，还可以动态地实现对请求内容的干预。但是，目前版本下的动态过滤器还是一个半成品，从 org.springframework.cloud.netflix.zuul.ZuulFilterInitializer 的源码中我们也可以看到，对于动态过滤器的加载是被注释掉的，并且被标注了 TODO 状态。不过，目前在实际使用过程中，对于处理一些简单的常用过滤功能还是没有什么问题的，只是需要注意一些已知的问题并避开这些情况来使用即可。比如，在使用 Groovy 定义动态过滤器的时候，删除 Groovy 文件并不能从当前运行的 API 网关中移除这个过滤器，所以如果要移除的话可以通过修改 Groovy 过滤器的 shouldFilter 返回 false。另外还需要注意一点，目前的动态过滤器是无法直接注入 API 网关服务的 Spring 容器中加载的实例来使用的，比如，我们是无法直接通过注入 RestTemplate 等实例，在动态过滤器中对各个微服务发起请求的。

第 8 章

分布式配置中心：Spring Cloud Config

Spring Cloud Config 是 Spring Cloud 团队创建的一个全新项目，用来为分布式系统中的基础设施和微服务应用提供集中化的外部配置支持，它分为服务端与客户端两个部分。其中服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置仓库并为客户端提供获取配置信息、加密/解密信息等访问接口；而客户端则是微服务架构中的各个微服务应用或基础设施，它们通过指定的配置中心来管理应用资源与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息。Spring Cloud Config 实现了对服务端和客户端中环境变量和属性配置的抽象映射，所以它除了适用于 Spring 构建的应用程序之外，也可以在任何其他语言运行的应用程序中使用。由于 Spring Cloud Config 实现的配置中心默认采用 Git 来存储配置信息，所以使用 Spring Cloud Config 构建的配置服务器，天然就支持对微服务应用配置信息的版本管理，并且可以通过 Git 客户端工具来方便地管理和访问配置内容。当然它也提供了对其他存储方式的支持，比如 SVN 仓库、本地化文件系统。接下来，我们从一个简单的入门示例开始学习 Spring Cloud Config 服务端以及客户端的详细构建与使用方法。

快速入门

在本节中，我们将演示如何构建一个基于 Git 存储的分布式配置中心，同时对配置的详细规则进行讲解，并在客户端中演示如何通过配置指定微服务应用的所属配置中心，并让其能够从配置中心获取配置信息并绑定到代码中的整个过程。

构建配置中心

通过 Spring Cloud Config 构建一个分布式配置中心非常简单，只需要以下三步。

- 创建一个基础的 Spring Boot 工程，命名为 config-server，并在 pom.xml 中引入下面的依赖：

```
<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>1.3.7.RELEASE</version>
   <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-config-server</artifactId>
   </dependency>
</dependencies>

<dependencyManagement>
   <dependencies>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-dependencies</artifactId>
         <version>Brixton.SR5</version>
         <type>pom</type>
         <scope>import</scope>
      </dependency>
   </dependencies>
</dependencyManagement>
```

- 创建 Spring Boot 的程序主类，并添加@EnableConfigServer 注解，开启 Spring Cloud Config 的服务端功能。

```
@EnableConfigServer
@SpringBootApplication
public class Application {

   public static void main(String[] args) {
      new SpringApplicationBuilder(Application.class).web(true).run(args);
   }

}
```

- 在 application.properties 中添加配置服务的基本信息以及 Git 仓库的相关信息，如下所示：

```
spring.application.name=config-server
server.port=7001

spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/SpringCloud
-Learning/
spring.cloud.config.server.git.searchPaths=spring_cloud_in_action/config-repo
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

其中 Git 的配置信息分别表示如下内容。

- spring.cloud.config.server.git.uri：配置 Git 仓库位置。
- spring.cloud.config.server.git.searchPaths：配置仓库路径下的相对搜索位置，可以配置多个。
- spring.cloud.config.server.git.username：访问 Git 仓库的用户名。
- spring.cloud.config.server.git.password：访问 Git 仓库的用户密码。

到这里，使用一个通过 Spring Cloud Config 实现，并使用 Git 管理配置内容的分布式配置中心就完成了。我们可以将该应用先启动起来，确保没有错误产生，然后进入下面的学习内容。

配置规则详解

为了验证上面完成的分布式配置中心 config-server，根据 Git 配置信息中指定的仓库位置，在 http://git.oschina.net/didispace/SpringCloud-Learning/spring_cloud_in_action/下创建了一个 config-repo 目录作为配置仓库，并根据不同环境新建下面 4 个配置文件：

- didispace.properties
- didispace-dev.properties
- didispace-test.properties
- didispace-prod.properties

在这 4 个配置文件中均设置了一个 from 属性，并为每个配置文件分别设置了不同的值，如下所示：

- from=git-default-1.0
- from=git-dev-1.0
- from=git-test-1.0

- from=git-prod-1.0

为了测试版本控制，在该 Git 仓库的 master 分支中，我们为 from 属性加入 1.0 的后缀，同时创建一个 config-label-test 分支，并将各配置文件中的值用 2.0 作为后缀。

完成了这些准备工作之后，我们就可以通过浏览器、POSTMAN 或 CURL 等工具直接来访问我们的配置内容了。访问配置信息的 URL 与配置文件的映射关系如下所示：

- /{application}/{profile}[/{label}]
- /{application}-{profile}.yml
- /{label}/{application}-{profile}.yml
- /{application}-{profile}.properties
- /{label}/{application}-{profile}.properties

上面的 url 会映射{application}-{profile}.properties 对应的配置文件，其中{label}对应 Git 上不同的分支，默认为 master。我们可以尝试构造不同的 url 来访问不同的配置内容，比如，要访问 config-label-test 分支，didispace 应用的 prod 环境，就可以访问这个 url：http://localhost:7001/didispace/prod/config-label-test，并获得如下返回信息：

```
{
    "name": "didispace",
    "profiles": [
        "prod"
    ],
    "label": "config-label-test",
    "version": "4c4f3909b0499b8518aba1f76e8a90b0dbad535d",
    "propertySources": [
        {
            "name": "http://git.oschina.net/didispace/SpringCloud-Learning/spring_
cloud_in_action/config-repo/didispace-prod.properties",
            "source": {
                "from": "git-prod-2.0"
            }
        },
        {
            "name": "http://git.oschina.net/didispace/SpringCloud-Learning/spring_
cloud_in_action/config-repo/didispace.properties",
            "source": {
                "from": "git-default-2.0"
            }
        }
    ]
}
```

我们可以看到该 JSON 中返回了应用名 didispace，环境名 prod，分支名 config-label-test，以及 default 环境和 prod 环境的配置内容。另外，之前没有提到过的 version，从下图我们可以观察到，它对应的是在 Git 上的 commit 号。

同时，我们可以看到 config-server 的控制台中还输出了下面的内容，配置服务器在从 Git 中获取配置信息后，会存储一份在 config-server 的文件系统中，实质上 config-server 是通过 git clone 命令将配置内容复制了一份在本地存储，然后读取这些内容并返回给微服务应用进行加载。

```
s.c.a.AnnotationConfigApplicationContext : Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@7d09c9:
startup date [Fri Sep 16 21:56:43 CST 2016]; root of context hierarchy
o.s.c.c.s.e.NativeEnvironmentRepository  : Adding property source:
file:/C:/Users/ADMINI~1/AppData/Local/Temp/config-repo-914347082723810766/spring_cl
oud_in_action/config-repo/didispace-prod.properties
o.s.c.c.s.e.NativeEnvironmentRepository  : Adding property source:
file:/C:/Users/ADMINI~1/AppData/Local/Temp/config-repo-914347082723810766/spring_cl
oud_in_action/config-repo/didispace.properties
s.c.a.AnnotationConfigApplicationContext : Closing
org.springframework.context.annotation.AnnotationConfigApplicationContext@7d09c9:
startup date [Fri Sep 16 21:56:43 CST 2016]; root of context hierarchy
```

config-server 通过 Git 在本地仓库暂存，可以有效防止当 Git 仓库出现故障而引起无法加载配置信息的情况。我们可以通过断开网络，再次发起 http://localhost:7001/didispace/prod/config-label-test 请求，在控制台中可输出如下内容。可以看到，config-server 提示无法从远程获取该分支内容的报错信息：Could not pull remote for config-label-test，但是它依然会为该请求返回配置内容，这些内容源于之前访问时存于 config-server 本地文件系统中的配置内容。

```
.c.s.e.MultipleJGitEnvironmentRepository : Could not pull remote for config-label-test
(current ref=Ref[refs/heads/config-label-test=
4c4f3909b0499b8518aba1f76e8a90b0dbad535d]), remote:
http://git.oschina.net/didispace/SpringCloud-Learning
s.c.a.AnnotationConfigApplicationContext : Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@55c747ad:
startup date [Mon Sep 19 10:51:04 CST 2016]; root of context hierarchy
o.s.c.c.s.e.NativeEnvironmentRepository  : Adding property source:
file:/C:/Users/ADMINI~1/AppData/Local/Temp/config-repo-7514536618307405051/spring_c
loud_in_action/config-repo/didispace-prod.properties
o.s.c.c.s.e.NativeEnvironmentRepository  : Adding property source:
file:/C:/Users/ADMINI~1/AppData/Local/Temp/config-repo-7514536618307405051/spring_c
loud_in_action/config-repo/didispace.properties
s.c.a.AnnotationConfigApplicationContext : Closing
```

```
org.springframework.context.annotation.AnnotationConfigApplicationContext@55c747ad:
startup date [Mon Sep 19 10:51:04 CST 2016]; root of context hierarchy
```

客户端配置映射

在完成了上述验证之后，确定配置服务中心已经正常运作，下面我们尝试如何在微服务应用中获取上述配置信息。

- 创建一个 Spring Boot 应用，命名为 `config-client`，并在 `pom.xml` 中引入下述依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- 创建 Spring Boot 的应用主类，具体如下：

```
@SpringBootApplication
public class Application {
```

```
    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

- 创建 bootstrap.properties 配置，来指定获取配置文件的 config-server 位置，例如：

```
spring.application.name=didispace
spring.cloud.config.profile=dev
spring.cloud.config.label=master
spring.cloud.config.uri=http://localhost:7001/

server.port=7002
```

上述配置参数与 Git 中存储的配置文件中各个部分的对应关系如下所示。

- spring.application.name：对应配置文件规则中的{application}部分。
- spring.cloud.config.profile：对应配置文件规则中的{profile}部分。
- spring.cloud.config.label：对应配置文件规则中的{label}部分。
- spring.cloud.config.uri：配置中心 config-server 的地址。

这里需要格外注意，上面这些属性必须配置在 bootstrap.properties 中，这样 config-server 中的配置信息才能被正确加载。在第 2 章中，我们详细说明了 Spring Boot 对配置文件的加载顺序，对于本应用 jar 包之外的配置文件加载会优先于应用 jar 包内的配置内容，而通过 bootstrap.properties 对 config-server 的配置，使得该应用会从 config-server 中获取一些外部配置信息，这些信息的优先级比本地的内容要高，从而实现了外部化配置。

- 创建一个 RESTful 接口来返回配置中心的 from 属性，通过@Value("${from}") 绑定配置服务中配置的 from 属性，具体实现如下：

```
@RefreshScope
@RestController
public class TestController {

    @Value("${from}")
    private String from;

    @RequestMapping("/from")
    public String from() {
        return this.from;
    }
}
```

- 除了通过@Value 注解绑定注入之外，也可以通过 Environment 对象来获取配置属性，比如：

```
@RefreshScope
@RestController
public class TestController {

    @Autowired
    private Environment env;

    @RequestMapping("/from")
    public String from() {
        return env.getProperty("from", "undefined");
    }
}
```

启动 config-client 应用，并访问 http://localhost:7002/from，我们就可以根据配置内容输出对应环境的 from 内容了。根据当前配置，我们可以获得如下返回内容：git-dev-1.0。可以继续通过修改 bootstrap.properties 中的配置内容获取不同的配置信息来熟悉配置服务中的配置规则。

服务端详解

在上一节中，我们实现了一个具备基本结构的配置管理服务端和客户端，同时讲解了其中一些配置的基本原理和规则。在本节中，我们将进一步介绍 Spring Cloud Config 服务端的一些相关知识和用法。

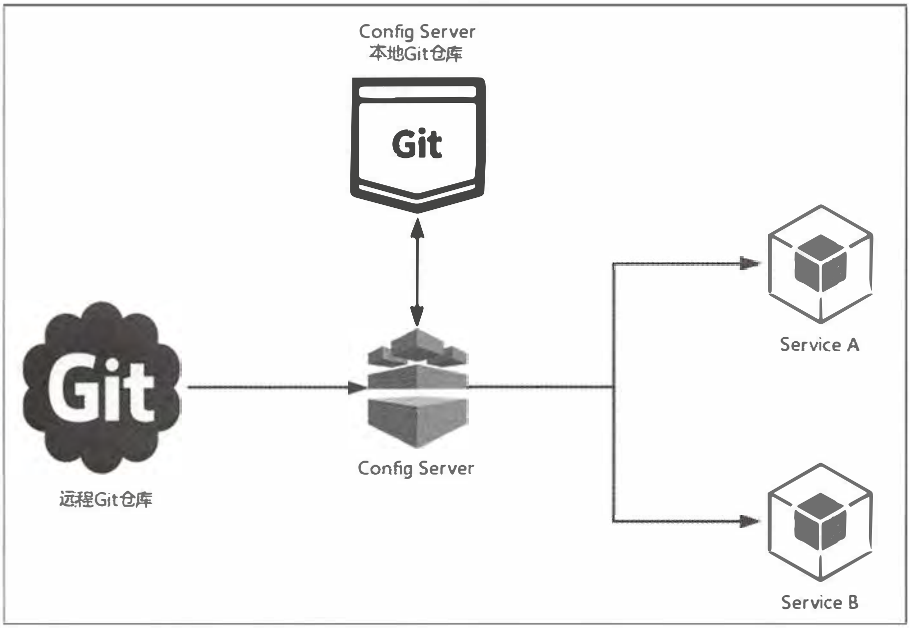
基础架构

在动手实践了上面关于 Spring Cloud Config 的基础入门内容之后，在这里我们深入理解一下它是如何运作起来的。下图所示的是上一节我们所构建案例的基本结构。

其中，主要包含下面几个要素。

- 远程 Git 仓库：用来存储配置文件的地方，上例中我们用来存储针对应用名为 didispace 的多环境配置文件：didispace-{profile}.properties。
- Config Server：这是我们上面构建的分布式配置中心，config-server 工程，在该工程中指定了所要连接的 Git 仓库位置以及账户、密码等连接信息。
- 本地 Git 仓库：在 Config Server 的文件系统中，每次客户端请求获取配置信息时，Config Server 从 Git 仓库中获取最新配置到本地，然后在本地 Git 仓库中读取并返回。当远程仓库无法获取时，直接将本地内容返回。

- Service A、Service B：具体的微服务应用，它们指定了 Config Server 的地址，从而实现从外部化获取应用自己要用的配置信息。这些应用在启动的时候，会向 Config Server 请求获取配置信息来进行加载。



客户端应用从配置管理中获取配置信息遵从下面的执行流程：

1. 应用启动时，根据 `bootstrap.properties` 中配置的应用名`{application}`、环境名`{profile}`、分支名`{label}`，向 Config Server 请求获取配置信息。
2. Config Server 根据自己维护的 Git 仓库信息和客户端传递过来的配置定位信息去查找配置信息。
3. 通过 `git clone` 命令将找到的配置信息下载到 Config Server 的文件系统中。
4. Config Server 创建 Spring 的 `ApplicationContext` 实例，并从 Git 本地仓库中加载配置文件，最后将这些配置内容读取出来返回给客户端应用。
5. 客户端应用在获得外部配置文件后加载到客户端的 `ApplicationContext` 实例，该配置内容的优先级高于客户端 Jar 包内部的配置内容，所以在 Jar 包中重复的内容将不再被加载。

Config Server 巧妙地通过 `git clone` 将配置信息存于本地，起到了缓存的作用，即使当 Git 服务端无法访问的时候，依然可以取 Config Server 中的缓存内容进行使用。

Git 配置仓库

在 Spring Cloud Config 的服务端，对于配置仓库的默认实现采用了 Git。Git 非常适用于存储配置内容，它可以非常方便地使用各种第三方工具来对其内容进行管理更新和版本化，同时 Git 仓库的 Hook 功能还可以帮助我们实时地监控配置内容的修改。其中，Git 自身的版本控制功能正是其他一些配置中心所欠缺的，通过 Git 进行存储意味着，一个应用的不同部署实例可以从 Spring Cloud Config 的服务端获取不同的版本配置，从而支持一些特殊的应用场景。

由于 Spring Cloud Config 中默认使用 Git，所以对于 Git 的配置也非常简单，只需在 Config Server 的 `application.properties` 中设置 `spring.cloud.config.server.git.uri` 属性，为其指定 Git 仓库的网络地址和账户信息即可，比如在快速入门一节中的例子：

```
spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/SpringCloud
-Learning/
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

如果我们将该值通过 `file://`前缀来设置为一个文件地址（在 Windows 系统中，需要使用 `file:///`来定位文件内容），那么它将以本地仓库的方式运行，这样我们就可以脱离 Git 服务端来快速进行调试与开发，比如：

```
spring.cloud.config.server.git.uri=file://${user.home}/config-repo
```

其中，`${user.home}`代表当前用户的所属目录。`file://`配置的本地文件系统方式虽然对于本地开发调试时使用非常方便，但是该方式也仅用于开发与测试，在生产环境中请务必搭建自己的 Git 仓库来存储配置资源。

占位符配置 URI

`{application}`、`{profile}`、`{label}`这些占位符除了用于标识配置文件的规则之外，还可以用于 Config Server 中对 Git 仓库地址的 URI 配置。比如，我们可以通过 `{application}`占位符来实现一个应用对应一个 Git 仓库目录的配置效果，具体配置实现如下：

```
spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/{application}
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

其中，`{application}`代表了应用名，所以当客户端应用向 Config Server 发起获取配置的请求时，Config Server 会根据客户端的 `spring.application.name` 信息来填充 `{application}`占位符以定位配置资源的存储位置，从而实现根据微服务应用的属性动

态获取不同位置的配置。另外，在这些占位符中，{label}参数较为特别，如果 Git 的分支和标签名包含“/”，那么{label}参数在 HTTP 的 URL 中应该使用“(_)”来替代，以避免改变了 URI 含义，指向到其他的 URI 资源。

当我们使用 Git 作为配置中心来存储各个微服务应用配置文件的时候，该功能会变得非常有用，通过在 URI 中使用占位符可以帮助我们规划和实现通用的仓库配置。例如，我们可以对微服务应用做如下规划。

- **代码库**：使用服务名作为 Git 仓库名称，比如会员服务的代码库 http://git.oschina.net/didispace/member-service。
- **配置库**：使用服务名加上-config 后缀作为 Git 仓库名称，比如上面会员服务对应的配置库地址位置 http://git.oschina.net/didispace/member-service-config。

这时，我们就可以使用 spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/{application}-config 配置，来同时匹配多个不同服务的配置仓库。

配置多个仓库

Config Server 除了可以通过 application 和 profile 模式来匹配配置仓库之外，还支持通过带有通配符的表达式来匹配，以实现更为复杂的配置需求。并且当我们有多个匹配规则的时候，还可以用逗号来分割多个{application}/{profile}配置规则，比如：

```
spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/config-repo

spring.cloud.config.server.git.repos.dev.pattern=dev/*
spring.cloud.config.server.git.repos.dev.uri=file://home/git/config-repo

spring.cloud.config.server.git.repos.test=http://git.oschina.net/test/config-repo

spring.cloud.config.server.git.repos.prod.pattern=prod/pp*,online/oo*
spring.cloud.config.server.git.repos.prod.uri=http://git.oschina.net/prod/confi
g-repo
```

上述配置内容通过 spring.cloud.config.server.git.uri 属性，指定了一个默认的仓库位置，当使用{application}/{profile}模式未能匹配到合适的仓库时，就将在该默认仓库位置下获取配置信息。除此之外，还配置了三个仓库，分别是 dev、test、prod。其中，dev 仓库匹配 dev/*的模式，所以无论 profile 是什么，它都能匹配 application 名称为 dev 的应用。并且我们可以注意到，它存储的配置文件位置还采用

了 Config Server 的本地文件系统中的内容。对于此配置，我们可以通过访问 http://localhost:7001/dev/profile 的请求来验证到该仓库的配置内容，其中 profile 可以为任意值。而 test 和 prod 仓库均使用了 Git 仓库的存储，并且 test 仓库未配置匹配规则，所以它只匹配 application 名为 test 的应用；prod 仓库则需要匹配 application 为 prod 并且 profile 为 pp 开头，或者 application 为 online 并且 profile 为 oo 开头的应用和环境。

当配置多个仓库的时候，Config Server 在启动时会直接克隆第一个仓库的配置库，其他的配置库只有在请求时才会克隆到本地，所以对于仓库的排列可以根据配置内容的重要程度有所区分。另外，如果表达式是以通配符开始的，那么需要使用引号将配置内容引起来。

子目录存储

除了支持占位符配置、多仓库配置之外，Config Server 还可以将配置文件定位到 Git 仓库的子目录中。有心的读者，或许还能记得在快速入门中，我们除了配置 spring.cloud.config.server.git.uri 属性之外，还配置了另外一个参数：spring.cloud.config.server.git.searchPaths，通过该参数我们实现了在 http://git.oschina.net/didispace/SpringCloud-Learning/仓库的 spring_cloud_in_action/config-repo 子目录下实现配置的存储。

对于 spring.cloud.config.server.git.searchPaths 参数的配置也支持使用{application}、{profile}和{label}占位符，比如：

```
    spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/SpringCloud
-Learning/
    spring.cloud.config.server.git.searchPaths={application}
```

通过上面的配置，我们可以实现在 http://git.oschina.net/didispace/SpringCloud-Learning/仓库中，一个应用一个目录的效果。

访问权限

Config Server 在访问 Git 仓库的时候，若采用 HTTP 的方式进行认证，那么我们需要增加 username 和 password 属性来配置账户（快速入门中也是如此实现），比如：

```
    spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/application
    spring.cloud.config.server.git.username=username
    spring.cloud.config.server.git.password=password
```

若不使用 HTTP 的认证方式，我们也可采用 SSH 的方式，通过生成 Key 并在 Git 仓库中进行配置匹配以实现访问。

SVN 配置仓库

Config Server 除了支持 Git 仓库之外，也能使用 SVN 仓库，只需要做如下配置。

- 在 pom.xml 中引入 SVN 的依赖配置，让 Config Server 拥有读取 SVN 内容的能力：

```
<dependency>
    <groupId>org.tmatesoft.svnkit</groupId>
    <artifactId>svnkit</artifactId>
    <version>1.8.10</version>
</dependency>
```

- 在 application.properties 中使用 SVN 的配置属性来指定 SVN 服务器的位置，以及访问的账户名与密码：

```
spring.cloud.config.server.svn.uri=svn://localhost:443/didispace/config-repo
spring.cloud.config.server.svn.username=username
spring.cloud.config.server.svn.password=password
```

通过上面的配置修改，Config Server 就可以使用 SVN 作为仓库来存储配置文件了，而对于客户端来说，这个过程是透明的，所以不需要做任何变动。

本地仓库

在使用了 Git 或 SVN 仓库之后，文件都会在 Config Server 的本地文件系统中存储一份，这些文件默认会被存储于以 config-repo 为前缀的临时目录中，比如名为 /tmp/config-repo-<随机数>的目录。由于其随机性以及临时目录的特性，可能会有一些不可预知的后果，为了避免将来可能会出现的问题，最好的办法就是指定一个固定的位置来存储这些重要信息。我们只需要通过 spring.cloud.config.server.git.basedir 或 spring.cloud.config.server.svn.basedir 来配置一个我们准备好的目录即可。

本地文件系统

Spring Cloud Config 也提供了一种不使用 Git 仓库或 SVN 仓库的存储方式，而是使用本地文件系统的存储方式来保存配置信息。实现方式也非常简单，我们只需要设置属性 spring.profiles.active=native，Config Server 会默认从应用的 src/main/resource 目录下搜索配置文件。如果需要指定搜索配置文件的路径，我们可以通过 spring.cloud.config.server.native.searchLocations 属性来指定具体的配置文件位置。

> 虽然 Spring Cloud Config 提供了这样的功能，但是为了支持更好的内容管理和版本控制等强大功能，还是推荐使用 Git 仓库的方式。

健康监测

当使用占位符来配置 URI 的时候，很有可能在控制台中出现类似这样的警告信息（以 spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/{application}-config 配置为例）：

```
    o.s.c.c.s.e.JGitEnvironmentRepository    : Could not fetch remote for master remote:
http://git.oschina.net/didispace/app-config
```

而引起该警告的原因是由于 Spring Cloud Config 的服务端为 spring-boot-actuator 模块的/health 端点实现了对应的健康检测器：org.springframework.cloud.config.server.config.ConfigServerHealthIndicator。在该检测器中，默认构建了一个 application 为 app 的仓库。而根据之前的配置规则：{application}-config.git，该检测器会不断地检查 http://git.oschina.net/didispace/app-config 仓库是否可以连通。此时，我们可以访问配置中心的/health 端点来查看它的健康状态，具体返回信息如下：

```
    "configServer": {
        "status": "DOWN",
        "repository": {
            "application": "app",
            "profiles": "default"
        },
        "error": "org.springframework.cloud.config.server.environment.NoSuchLabelException:
No such label: master"
    }
```

从/health 端点的返回信息中，我们可以看到，由于无法连通 http://git.oschina.net/didispace/app-config 仓库，使得配置中心的可用状态是 DOWN。虽然我们依然可以通过 URI 的方式访问该配置中心，但是当将配置中心服务化使用的时候，该状态将影响它的服务可用性判断。所以，我们需要了解它的健康检测配置，并让它的健康检测器正常工作起来。

根据默认配置规则，我们可以直接在 Git 仓库中创建一个名为 app-config 的配置库，让健康检测器能够访问到它。另外，也可以配置一个实际存在的仓库来进行连通检测，比如下面的配置，它实现了通过连接 check-repo-config 仓库来进行健康监测：

```
    spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/{applicatio
n}-config
    spring.cloud.config.server.git.username=username
    spring.cloud.config.server.git.username=password
    spring.cloud.config.server.health.repositories.check.name=check-repo
    spring.cloud.config.server.health.repositories.check.label=master
    spring.cloud.config.server.health.repositories.check.profiles=default
```

由于健康检测的 repositories 是个 Map 对象，所以实际使用时我们可以配置多个。而每个配置中包含了与定位仓库地址时类似的三个元素。

- name：应用名。
- label：分支名。
- profiles：环境名。

如果我们不想使用该健康检测器，也可以通过使用 spring.cloud.config.server.health.enabled=false 参数设置来关闭它。

属性覆盖

Config Server 还有一个“属性覆盖”的特性，它可以让开发人员为所有的应用提供配置属性，只需要通过 spring.cloud.config.server.overrides 属性来设置键值对的参数，这些参数会以 Map 的方式加载到客户端的配置中。比如：

```
spring.cloud.config.server.overrides.name=didi
spring.cloud.config.server.overrides.from=shanghai
```

通过该属性配置的参数，不会被 Spring Cloud 的客户端修改，并且 Spring Cloud 客户端从 Config Server 中获取配置信息时，都会取得这些配置信息。利用该特性可以方便地为 Spring Cloud 应用配置一些共同属性或是默认属性。当然，这些属性并非强制的，我们可以通过改变客户端中更高优先级的配置方式（比如，配置环境变量或是系统属性），来选择是否使用 Config Server 提供的默认值。

安全保护

由于配置中心存储的内容比较敏感，做一定的安全处理是必需的。为配置中心实现安全保护的方式有很多，比如物理网络限制、OAuth2 授权等。不过，由于我们的微服务应用和配置中心都构建于 Spring Boot 基础上，所以与 Spring Security 结合使用会更加方便。

我们只需要在配置中心的 pom.xml 中加入 spring-boot-starter-security 依赖，不需要做任何其他改动就能实现对配置中心访问的安全保护。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

默认情况下，我们可以获得一个名为 user 的用户，并且在配置中心启动的时候，在日志中打印出该用户的随机密码，具体如下：

```
INFO 22028 --- [           main] b.a.s.AuthenticationManagerConfiguration : Using
default security password: 1a32a848-da0c-4590-9c58-e860be8c50dd
```

大多数情况下，我们并不会使用随机生成密码的机制。我们可以在配置文件中指定用户和密码，比如：

```
security.user.name=user
security.user.password=37cc5635-559b-4e6f-b633-7e932b813f73
```

由于我们已经为 config-server 设置了安全保护，如果这时候连接到配置中心的客户端中没有设置对应的安全信息，在获取配置信息时会返回 401 错误。所以，需要通过配置的方式在客户端中加入安全信息来通过校验，比如：

```
spring.cloud.config.username=user
spring.cloud.config.password=37cc5635-559b-4e6f-b633-7e932b813f73
```

加密解密

在微服务架构中，我们通常会采用 DevOps 的组织方式来降低因团队间沟通造成的巨大成本，以加速微服务应用的交付能力。这就使得原本由运维团队控制的线上信息将交由微服务所属组织的成员自行维护，其中将会包括大量的敏感信息，比如数据库的账户与密码等。显然，如果我们直接将敏感信息以明文的方式存储于微服务应用的配置文件中是非常危险的。针对这个问题，Spring Cloud Config 提供了对属性进行加密解密的功能，以保护配置文件中的信息安全。比如下面的例子：

```
spring.datasource.username=didi
spring.datasource.password={cipher}dba6505baa81d78bd08799d8d4429de499bd4c2053c0
5f029e7cfbf143695f5b
```

在 Spring Cloud Config 中通过在属性值前使用{cipher}前缀来标注该内容是一个加密值，当微服务客户端加载配置时，配置中心会自动为带有{cipher}前缀的值进行解密。通过该机制的实现，运维团队就可以放心地将线上信息的加密资源给到微服务团队，而不用担心这些敏感信息遭到泄露了。下面我们来具体介绍如何在配置中心使用该项功能。

使用前提

在使用 Spring Cloud Config 的加密解密功能时，有一个必要的前提需要我们注意。为了启用该功能，我们需要在配置中心的运行环境中安装不限长度的 JCE 版本（Unlimited Strength Java Cryptography Extension）。虽然，JCE 功能在 JRE 中自带，但是默认使用的是有长度限制的版本。我们可以从 Oracle 的官方网站下载到它，它是一个压缩包，解压后可以看到下面三个文件：

```
README.txt
local_policy.jar
US_export_policy.jar
```

我们需要将 local_policy.jar 和 US_export_policy.jar 两个文件复制到

$JAVA_HOME/jre/lib/security 目录下，覆盖原来的默认内容。到这里，加密解密的准备工作就完成了。

相关端点

在完成了 JCE 的安装后，可以尝试启动配置中心。在控制台中，将会输出一些配置中心特有的端点，主要包括如下几个。

- /encrypt/status：查看加密功能状态的端点。
- /key：查看密钥的端点。
- /encrypt：对请求的 body 内容进行加密的端点。
- /decrypt：对请求的 body 内容进行解密的端点。

可以尝试通过 GET 请求访问/encrypt/status 端点，我们将得到如下内容：

```
{
  "description": "No key was installed for encryption service",
  "status": "NO_KEY"
}
```

该返回信息说明当前配置中心的加密功能还不能使用，因为没有为加密服务配置对应的密钥。

配置密钥

我们可以通过 encrypt.key 属性在配置文件中直接指定密钥信息（对称性密钥），比如：

```
encrypt.key=didispace
```

加入上述配置信息后，重启配置中心，再访问/encrypt/status 端点，我们将得到如下内容：

```
{
  "status": "OK"
}
```

此时，我们配置中心的加密解密功能就已经可以使用了，不妨尝试访问一下/encrypt 和/decrypt 端点来使用加密和解密的功能。注意，这两个端点都是 POST 请求，加密和解密信息需要通过请求体来发送。比如，以 curl 命令为例，我们可以通过下面的方式调用加密与解密端点：

```
$ curl localhost:7001/encrypt -d didispace
3c70a809bfa24ab88bcb5e1df51cb9e4dd4b8fec88301eb7a18177f1769c849ae9c9f29400c9204
80be2c99406ae28c7

$ curl localhost:7001/decrypt -d
3c70a809bfa24ab88bcb5e1df51cb9e4dd4b8fec88301eb7a18177f1769c849ae9c9f29400c920480be
```

```
2c99406ae28c7
    didispace
```

这里，我们通过配置 `encrypt.key` 参数来指定密钥的实现方式采用了对称性加密。这种方式实现起来比较简单，只需要配置一个参数即可。另外，我们也可以使用环境变量 `ENCRYPT_KEY` 来进行配置，让密钥信息外部化存储。

非对称加密

Spring Cloud Config 的配置中心不仅可以使用对称性加密，也可以使用非对称性加密（比如 RSA 密钥对）。虽然非对称性加密的密钥生成与配置相对复杂一些，但是它具有更高的安全性。下面，我们来具体介绍一下如何使用非对称加密。

首先，需要通过 `keytool` 工具来生成密钥对。`keytool` 是 JDK 中的一个密钥和证书管理工具。它使用户能够管理自己的公钥/私钥对及相关证书，用于（通过数字签名）自我认证（用户向其他用户/ 服务认证自己）或数据完整性以及认证服务。在 JDK 1.4 以后的版本中都包含了这一工具，它的位置在`%JAVA_HOME%\bin\keytool.exe`。

生成密钥的具体命令如下所示：

```
    $ keytool -genkeypair -alias config-server -keyalg RSA -keystore
config-server.keystore
    输入密钥库口令:
    再次输入新口令:
    您的名字与姓氏是什么?
      [Unknown]:  zhaiyongchao
    您的组织单位名称是什么?
      [Unknown]:  company
    您的组织名称是什么?
      [Unknown]:  organization
    您所在的城市或区域名称是什么?
      [Unknown]:  city
    您所在的省/市/自治区名称是什么?
      [Unknown]:  province
    该单位的双字母国家/地区代码是什么?
      [Unknown]:  china
    CN=zhaiyongchao, OU=company, O=organization, L=city, ST=province, C=china 是否正确?
      [否]:  y

    输入 <config-server> 的密钥口令
            (如果和密钥库口令相同, 按回车):
    再次输入新口令:
```

另外，如果不想逐步输入那些提示信息，可以使用-dname 来直接指定，而密钥库口令与密钥口令可使用-storepass 和-keypass 来直接指定。所以，我们可以通过下面的命令直接创建出与上述命令一样的密钥库：

```
$ keytool -genkeypair -alias config-server -keyalg RSA \
  -dname "CN=zhaiyongchao, OU=company, O=organization, L=city, ST=province, C=china" \
  -keypass 222222 \
  -keystore config-server.keystore \
  -storepass 111111 \
```

默认情况下，使用上述命令创建的密钥只有 90 天有效期。如果想要调整它的有效期，可以通过增加-validity 参数来实现，比如我们可以通过下面的命令，让密钥的有效期延长到一年：

```
$ keytool -genkeypair -alias config-server -keyalg RSA \
  -dname "CN=zhaiyongchao, OU=company, O=organization, L=city, ST=province, C=china" \
  -keypass 222222 \
  -keystore config-server.keystore \
  -storepass 111111 \
  -validity 365 \
```

上述的三种命令生成方式，最终都会在命令的当前执行目录下生成一个 config-server.keystore 文件。下面，我们需要将它保存在配置中心的文件系统中的某个位置，比如放在当前的用户目录下，然后在配置中心中加入相关的配置信息：

```
encrypt.key-store.location=file://${user.home}/config-server.keystore
encrypt.key-store.alias=config-server
encrypt.key-store.password=111111
encrypt.key-store.secret=222222
```

如果我们将 config-server.keystore 放在配置中心的 src/main/resource 目录下，也可以直接这样配置：encrypt.key-store.location=config-server.keystore。另外，非对称加密的配置信息也可以通过环境变量的方式进行配置，它们对应的具体变量名如下：

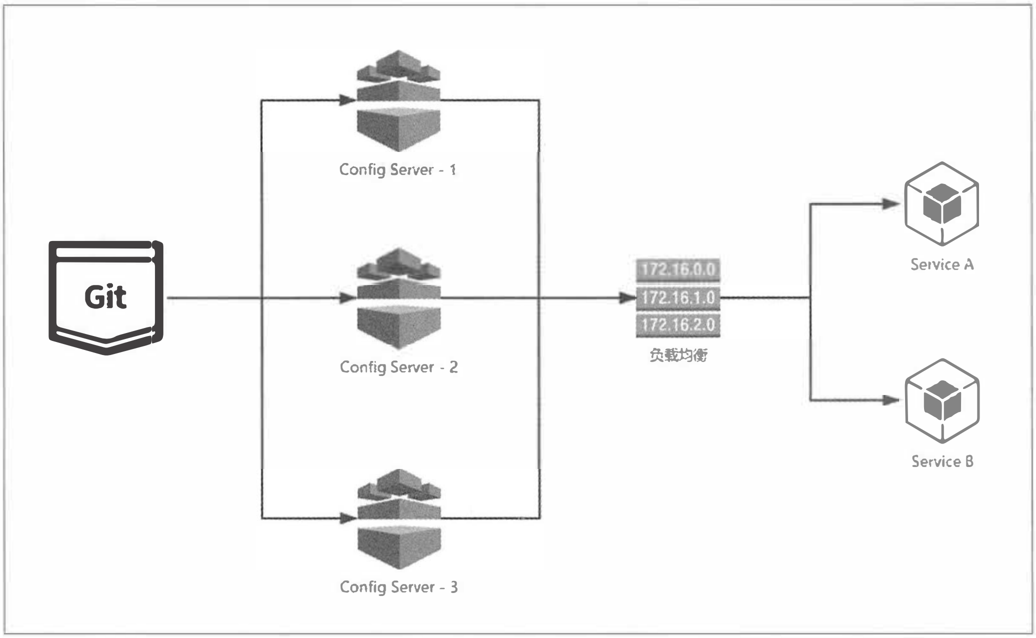
```
ENCRYPT_KEY_STORE_LOCATION
ENCRYPT_KEY_STORE_ALIAS
ENCRYPT_KEY_STORE_PASSWORD
ENCRYPT_KEY_STORE_SECRET
```

通过环境变量来配置密钥库相关信息可以获得更好的安全性，所以我们将敏感的口令信息存储在配置中心的环境变量中是一种不错的选择。

高可用配置

当要将配置中心部署到生产环境中时，与服务注册中心一样，我们也希望它是一个高可用的应用。Spring Cloud Config 实现服务端的高可用非常简单，主要有以下两种方式。

- 传统模式：不需要为这些服务端做任何额外的配置，只需要遵守一个配置规则，将所有的 Config Server 都指向同一个 Git 仓库，这样所有的配置内容就通过统一的共享文件系统来维护。而客户端在指定 Config Server 位置时，只需要配置 Config Server 上层的负载均衡设备地址即可，就如下图所示的结构。



- 服务模式：除了上面这种传统的实现模式之外，我们也可以将 Config Server 作为一个普通的微服务应用，纳入 Eureka 的服务治理体系中。这样我们的微服务应用就可以通过配置中心的服务名来获取配置信息，这种方式比起传统的实现模式来说更加有利于维护，因为对于服务端的负载均衡配置和客户端的配置中心指定都通过服务治理机制一并解决了，既实现了高可用，也实现了自维护。由于这部分的实现需要客户端的配合，具体示例读者可详细阅读“客户端详解”一节中的“服务化配置中心”小节。

客户端详解

在学习了关于 Spring Cloud Config 服务端的大量配置和使用细节之后，接下来我们将

通过本节内容继续学习 Spring Cloud Config 客户端的使用与配置。

URI 指定配置中心

Spring Cloud Config 的客户端在启动的时候，默认会从工程的 classpath 中加载配置信息并启动应用。只有当我们配置 `spring.cloud.config.uri` 的时候，客户端应用才会尝试连接 Spring Cloud Config 的服务端来获取远程配置信息并初始化 Spring 环境配置。同时，我们必须将该参数配置在 `bootstrap.properties`、环境变量或是其他优先级高于应用 Jar 包内的配置信息中，才能正确加载到远程配置。若不指定 `spring.cloud.config.uri` 参数的话，Spring Cloud Config 的客户端会默认尝试连接 `http://localhost:8888`。

在之前的快速入门示例中，我们就是以这种方式实现的，就如下面的配置，其中 `spring.application.name` 和 `spring.cloud.config.profile` 用于定位配置信息。

```
spring.application.name=didispace
spring.cloud.config.profile=dev

spring.cloud.config.uri=http://localhost:7001/
```

服务化配置中心

在第 3 章中，我们已经学会了如何构建服务注册中心、如何发现与注册服务。那么 Config Server 是否也能以服务的方式注册到服务中心，并被其他应用所发现来实现配置信息的获取呢？答案是肯定的。在 Spring Cloud 中，我们也可以把 Config Server 视为微服务架构中与其他业务服务一样的一个基本单元。

下面，我们就来详细介绍如何将 Config Server 注册到服务中心，并通过服务发现来访问 Config Server 并获取 Git 仓库中的配置信息。下面的内容将基于快速入门中实现的 `config-server` 和 `config-client` 工程来进行改造实现。

服务端配置

- 在 `config-server` 的 `pom.xml` 中增加 `spring-cloud-starter-eureka` 依赖，以实现将分布式配置中心加入 Eureka 的服务治理体系。

```
......
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
    <dependency>
```

```
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
</dependencies>
......
```

- 在 application.properties 中配置参数 eureka.client.serviceUrl.defaultZone 以指定服务注册中心的位置，详细内容如下：

```
spring.application.name=config-server
server.port=7001
# 配置服务注册中心
eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
# Git 管理配置
spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/SpringCloud
-Learning/
spring.cloud.config.server.git.searchPaths=spring_cloud_in_action/config-repo
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

- 在应用主类中，新增@EnableDiscoveryClient 注解，用来将 config-server 注册到上面配置的服务注册中心上去。

```
@EnableDiscoveryClient
@EnableConfigServer
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

- 启动该应用，并访问 http://localhost:1111/，可以在 Eureka Server 的信息面板中看到 config-server 已经被注册了。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIG-SERVER	n/a (1)	(1)	UP (1) - PC-201602152056:config-server:7001

客户端配置

- 在 config-client 的 pom.xml 中新增 spring-cloud-starter-eureka 依

赖，以实现客户端发现 config-server 服务，具体配置如下：

```
……
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
</dependencies>
……
```

- 在 bootstrap.properties 中，按如下配置：

```
spring.application.name=didispace
server.port=7002

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/

spring.cloud.config.discovery.enabled=true
spring.cloud.config.discovery.serviceId=config-server
spring.cloud.config.profile=dev
```

其中，通过 eureka.client.serviceUrl.defaultZone 参数指定服务注册中心，用于服务的注册与发现；再将 spring.cloud.config.discovery.enabled 参数设置为 true，开启通过服务来访问 Config Server 的功能；最后利用 spring.cloud.config.discovery.serviceId 参数来指定 Config Server 注册的服务名。这里的 spring.application.name 和 spring.cloud.config.profile 如之前通过 URI 的方式访问的时候一样，用来定位 Git 中的资源。

- 在应用主类中，增加@EnableDiscoveryClient 注解，用来发现 config-server 服务，利用其来加载应用配置：

```
@EnableDiscoveryClient
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
```

```
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

- 沿用之前我们创建的 Controller 来加载 Git 中的配置信息：

```
@RefreshScope
@RestController
public class TestController {

    @Value("${from}")
    private String from;

    @RequestMapping("/from")
    public String from() {
        return this.from;
    }

}
```

- 完成上述配置之后，我们启动该客户端应用。若启动成功，访问 `http://localhost:1111/`，可以在 EurekaServer 的信息面板中看到该应用已经被注册成功。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIG-SERVER	n/a (1)	(1)	UP (1) - PC-201602152056:config-server:7001
DIDISPACE	n/a (1)	(1)	UP (1) - PC-201602152056:didispace:7002

- 访问客户端应用提供的服务 `http://localhost:7002/from`，此时，我们会返回在 Git 仓库中 `didispace-dev.properties` 文件中配置的 `from` 属性内容：`"git-dev-1.0"`。

失败快速响应与重试

SpringCloudConfig 的客户端会预先加载很多其他信息，然后再开始连接 Config Server 进行属性的注入。当我们构建的应用较为复杂的时候，可能在连接 Config Server 之前花费较长的启动时间，而在一些特殊场景下，我们又希望可以快速知道当前应用是否能顺利地从 Config Server 获取到配置信息，这对在初期构建调试环境时，可以减少很多等待启动的时间。要实现客户端优先判断 Config Server 获取是否正常，并快速响应失败内容，只需在 `bootstrap.properties` 中配置参数 `spring.cloud.config.failFast= true` 即可。

我们可以实验一下，在未配置该参数前，不启动 Config Server，直接启动客户端应用，可以获得下面的报错信息。同时，在报错之前，可以看到客户端应用已经加载了很多内容，比如 Controller 的请求等。

```
    org.springframework.beans.factory.BeanCreationException: Error creating bean with
name 'scopedTarget.testController': Injection of autowired dependencies failed;
```

加上 spring.cloud.config.failFast=true 参数之后，再启动客户端应用，可以获得下面的报错信息，并且前置的加载内容少了很多，这样通过该参数有效避免了当 Config Server 配置有误时，不需要多等待前置的一些加载时间，实现了快速返回失败信息。

```
    java.lang.IllegalStateException: Could not locate PropertySource and the fail fast
property is set, failing
```

上面，我们演示了当 Config Server 宕机或是客户端配置不正确导致连接不到而启动失败的情况，快速响应的配置可以发挥比较好的效果。但是，若只是因为网络波动等其他间歇性原因导致的问题，直接启动失败似乎代价有些高。所以，Config 客户端还提供了自动重试的功能，在开启重试功能前，先确保已经配置了 spring.cloud.config.failFast=true，再进行下面的操作。

- 在客户端的 pom.xml 中增加 spring-retry 和 spring-boot-starter-aop 依赖，具体如下：

```
<dependencies>
......
    <dependency>
        <groupId>org.springframework.retry</groupId>
        <artifactId>spring-retry</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
    </dependency>
......
</dependencies>
```

- 不需要再做其他任何配置，启动客户端应用，在控制台中可以看到如下内容。客户端在连接 Config Server 失败之后，会继续尝试，直到第 6 次失败后，才返回错误信息。通过这样的重试机制，可以避免一些间歇性问题引起的失败导致客户端应用无法启动的情况。

```
    c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at:
http://PC-201602152056:7001/
    c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at:
http://PC-201602152056:7001/
```

```
    c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at:
http://PC-201602152056:7001/
    c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at:
http://PC-201602152056:7001/
    c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at:
http://PC-201602152056:7001/
    c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at:
http://PC-201602152056:7001/
    o.s.boot.SpringApplication               : Application startup failed
    java.lang.IllegalStateException: Could not locate PropertySource and the fail fast
property is set, failing
```

若对默认的最大重试次数和重试间隔等设置不满意，还可以通过下面的参数进行调整。

- spring.cloud.config.retry.multiplier：初始重试间隔时间（单位为毫秒），默认为 1000 毫秒。
- spring.cloud.config.retry.initial-interval：下一间隔的乘数，默认为 1.1，所以当最初间隔是 1000 毫秒时，下一次失败后的间隔为 1100 毫秒。
- spring.cloud.config.retry.max-interval：最大间隔时间，默认为 2000 毫秒。
- spring.cloud.config.retry.max-attempts：最大重试次数，默认为 6 次。

获取远程配置

在入门示例中，我们对{application}、{profile}、{label}这些参数已经有了一定的了解。在 Git 仓库中，一个形如{application}-{profile}.properties 或{application}-{profile}.yml 的配置文件，通过 URL 请求和客户端配置的访问对应可以总结如下。

- 通过向 Config Server 发送 GET 请求以直接的方式获取，可用下面的链接形式。
  - 不带{label}分支信息，默认访问 master 分支，可使用：
    - /{application}-{profile}.yml
    - /{application}-{profile}.properties
  - 带{label}分支信息，可使用：
    - /{label}/{application}-{profile}.yml
    - /{application}/{profile}[/{label}]
    - /{label}/{application}-{profile}.properties

- 通过客户端配置方式加载的内容如下所示。
  - spring.application.name：对应配置文件中的{application}内容。
  - spring.cloud.config.profile：对应配置文件中{profile}内容。
  - spring.cloud.config.label：对应分支内容，如不配置，默认为 master。

动态刷新配置

有时候，我们需要对配置内容做一些实时更新，那么 Spring Cloud Config 是否可以实现呢？答案显然是可以的。下面，我们以快速入门中的示例作为基础，看看如何进行改造来实现配置内容的实时更新。

首先，回顾一下，当前我们已经实现了哪些内容。

- config-repo：定义在 Git 仓库中的一个目录，其中存储了应用名为 didispace 的多环境配置文件，配置文件中有一个 from 参数。
- config-server：配置了 Git 仓库的服务端。
- config-client：指定了 config-server 为配置中心的客户端，应用名为 didispace，用来访问配置服务器以获取配置信息。该应用中提供了一个/from 接口，它会获取 config-repo/didispace-dev.properties 中的 from 属性返回。

在改造程序之前，我们先将 config-server 和 config-client 都启动起来，并访问客户端提供的 REST 接口 http://localhost:7002/from 来获取配置信息，获得的返回内容为 git-dev-1.0。接着，我们可以尝试使用 Git 工具修改当前配置的内容，比如，将 config-repo/didispace-dev.properties 中的 from 的值从 from=git-dev-1.0 修改为 from=git-dev-2.0，再访问 http://localhost:7002/from，可以看到其返回内容还是 git-dev-1.0。

接下来，我们将在 config-client 端做一些改造以实现配置信息的动态刷新。

- 在 config-client 的 pom.xml 中新增 spring-boot-starter-actuator 监控模块。其中包含了/refresh 端点的实现，该端点将用于实现客户端应用配置信息的重新获取与刷新。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- 重新启动 config-client，访问一次 http://localhost:7002/from，可以看到当前的配置值。
- 修改 Git 仓库 config-repo/didispace-dev.properties 文件中 from 的值。
- 再访问一次 http://localhost:7002/from，可以看到配置值没有改变。
- 通过 POST 请求发送到 http://localhost:7002/refresh，我们可以看到返回内容如下，代表 from 参数的配置内容被更新了：

```
[
  "from"
]
```

- 再访问一次 http://localhost:7002/from，可以看到配置值已经是更新后的值了。

通过上面的介绍，大家不难想到，该功能还可以同 Git 仓库的 Web Hook 功能进行关联，当有 Git 提交变化时，就给对应的配置主机发送/refresh 请求来实现配置信息的实时更新。但是，当我们的系统发展壮大之后，维护这样的刷新清单也将成为一个非常大的负担，而且很容易犯错，那么有什么办法可以解决这个复杂度呢？后续我们将介绍如何通过 Spring Cloud Bus 来实现以消息总线的方式进行配置变更的通知，并完成集群上的批量配置更新。

第 9 章

消息总线：Spring Cloud Bus

在微服务架构的系统中，我们通常会使用轻量级的消息代理来构建一个共用的消息主题让系统中所有微服务实例都连接上来，由于该主题中产生的消息会被所有实例监听和消费，所以我们称它为消息总线。在总线上的各个实例都可以方便地广播一些需要让其他连接在该主题上的实例都知道的消息，例如配置信息的变更或者其他一些管理操作等。

由于消息总线在微服务架构系统中被广泛使用，所以它同配置中心一样，几乎是微服务架构中的必备组件。Spring Cloud 作为微服务架构综合性的解决方案，对此自然也有自己的实现，这就是本章我们将要具体介绍的 Spring Cloud Bus。通过使用 Spring Cloud Bus，可以非常容易地搭建起消息总线，同时实现了一些消息总线中的常用功能，比如，配合 Spring Cloud Config 实现微服务应用配置信息的动态更新等。

在本章中，我们将从消息代理的基础开始，由浅入深地介绍如何使用 Spring Cloud Bus 构建微服务架构中的消息总线。

消息代理

消息代理（Message Broker）是一种消息验证、传输、路由的架构模式。它在应用程序之间起到通信调度并最小化应用之间的依赖的作用，使得应用程序可以高效地解耦通信过程。消息代理是一个中间件产品，它的核心是一个消息的路由程序，用来实现接收和分发消息，并根据设定好的消息处理流来转发给正确的应用。它包括独立的通信和消息传递协议，能够实现组织内部和组织间的网络通信。设计代理的目的就是为了能够从应用程序中传入消息，并执行一些特别的操作，下面这些是在企业应用中，我们经常需要使用消息代理的场景：

- 将消息路由到一个或多个目的地。

- 消息转化为其他的表现方式。
- 执行消息的聚集、消息的分解，并将结果发送到它们的目的地，然后重新组合响应返回给消息用户。
- 调用 Web 服务来检索数据。
- 响应事件或错误。
- 使用发布-订阅模式来提供内容或基于主题的消息路由。

目前已经有非常多的开源产品可以供大家使用，比如：

- ActiveMQ
- Kafka
- RabbitMQ
- RocketMQ
- ...

当前版本的 Spring Cloud Bus 仅支持两款中间件产品：RabbitMQ 和 Kafka。在下面的章节中，我们将分别介绍如何使用这两款消息中间件与 Spring Cloud Bus 配合实现消息总线。

RabbitMQ 实现消息总线

RabbitMQ 是实现了高级消息队列协议（AMQP）的开源消息代理软件，也称为面向消息的中间件。RabbitMQ 服务器是用高性能、可伸缩而闻名的 Erlang 语言编写而成的，其集群和故障转移是构建在开放电信平台框架上的。

> AMQP 是 Advanced Message Queuing Protocol 的简称，它是一个面向消息中间件的开放式标准应用层协议。它定义了以下这些特性：
>
> - 消息方向
> - 消息队列
> - 消息路由（包括点到点和发布-订阅模式）
> - 可靠性
> - 安全性
>
> AMQP 要求消息的提供者和客户端接收者的行为要实现对不同供应商可以用相同的方式（比如 SMTP、HTTP、FTP 等）进行互相操作。在以往的中间件标准中，主要还是建立在 API 级别，比如 JMS，集中于通过不同的中间件实现来建立标准化的程序间的互操作性，而不是在多个中间件产品间实现互操作性。

> AMQP 与 JMS 不同，JMS 定义了一个 API 和一组消息收发必须实现的行为，而 AMQP 是一个线路级协议。线路级协议描述的是通过网络发送的数据传输格式。因此，任何符合该数据格式的消息发送和接收工具都能互相兼容和进行操作，这样就能轻易实现跨技术平台的架构方案。

RabbitMQ 以 AMQP 协议实现，所以它可以支持多种操作系统、多种编程语言，几乎可以覆盖所有主流的企业级技术平台。在微服务架构消息中间件的选型中，它是一个非常适合且优秀的选择。因此，在 Spring Cloud Bus 中包含了对 Rabbit 的自动化默认配置，在下面的章节中，我们将先从 RabbitMQ 的基础安装和使用开始，循序渐进地学习如何与 Spring Cloud Bus 进行整合实现消息总线。

基本概念

在开始具体实践之前，我们先介绍一些关于 RabbitMQ 的基本概念，为后续的讲解做一些必要铺垫（如果对于 RabbitMQ 已经很熟悉的读者可以跳过本节，直接从“快速入门”一节开始阅读）。

- Broker：可以理解为消息队列服务器的实体，它是一个中间件应用，负责接收消息生产者的消息，然后将消息发送至消息接收者或者其他的 Broker。
- Exchange：消息交换机，是消息第一个到达的地方，消息通过它指定的路由规则，分发到不同的消息队列中去。
- Queue：消息队列，消息通过发送和路由之后最终到达的地方，到达 Queue 的消息即进入逻辑上等待消费的状态。每个消息都会被发送到一个或多个队列。
- Binding：绑定，它的作用就是把 Exchange 和 Queue 按照路由规则绑定起来，也就是 Exchange 和 Queue 之间的虚拟连接。
- Routing Key：路由关键字，Exchange 根据这个关键字进行消息投递。
- Virtual host：虚拟主机，它是对 Broker 的虚拟划分，将消费者、生产者和它们依赖的 AMQP 相关结构进行隔离，一般都是为了安全考虑。比如，我们可以在一个 Broker 中设置多个虚拟主机，对不同用户进行权限的分离。
- Connection：连接，代表生产者、消费者、Broker 之间进行通信的物理网络。
- Channel：消息通道，用于连接生产者和消费者的逻辑结构。在客户端的每个连接里，可建立多个 Channel，每个 Channel 代表一个会话任务，通过 Channel 可以隔离同一连接中的不同交互内容。
- Producer：消息生产者，制造消息并发送消息的程序。
- Consumer：消息消费者，接收消息并处理消息的程序。

消息投递到队列的整个过程大致如下：

1. 客户端连接到消息队列服务器，打开一个 Channel。
2. 客户端声明一个 Exchange，并设置相关属性。
3. 客户端声明一个 Queue，并设置相关属性。
4. 客户端使用 Routing Key，在 Exchange 和 Queue 之间建立好绑定关系。
5. 客户端投递消息到 Exchange。
6. Exchange 接收到消息后，根据消息的 Key 和已经设置的 Binding，进行消息路由，将消息投递到一个或多个 Queue 里。

Exchange 也有几种类型。

1. Direct 交换机：完全根据 Key 进行投递。比如，绑定时设置了 Routing Key 为 `abc`，那么客户端提交的消息，只有设置了 Key 为 `abc` 的才会被投递到队列。
2. Topic 交换机：对 Key 进行模式匹配后进行投递，可以使用符号`#`匹配一个或多个词，符号`*`匹配正好一个词。比如，`abc.#`匹配 `abc.def.ghi`，`abc.*`只匹配 `abc.def`。
3. Fanout 交换机：不需要任何 Key，它采取广播的模式，一个消息进来时，投递到与该交换机绑定的所有队列。

RabbitMQ 支持消息的持久化，也就是将数据写在磁盘上。为了数据安全考虑，大多数情况下都会选择持久化。消息队列持久化包括 3 个部分：

1. Exchange 持久化，在声明时指定 durable => 1。
2. Queue 持久化，在声明时指定 durable => 1。
3. 消息持久化，在投递时指定 delivery_mode => 2（1 是非持久化）。

如果 Exchange 和 Queue 都是持久化的，那么它们之间的 Binding 也是持久化的。如果 Exchange 和 Queue 两者之间有一个是持久化的，一个是非持久化的，就不允许建立绑定。

安装与使用

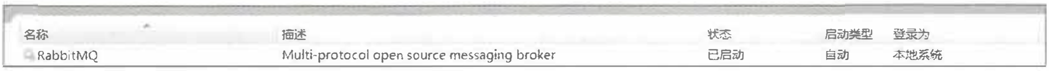
在 RabbitMQ 官网的下载页面 `https://www.rabbitmq.com/download.html` 中，我们可以获取到针对各种不同操作系统的安装包和说明文档。这里，我们将对几个常用的平台进行一一说明。

下面我们采用的是 Erlang 和 RabbitMQ Server 版本说明：

- Erlang/OTP 19.1
- RabbitMQ Server 3.6.5

在 Windows 系统中安装

1. 安装 Erlang，通过官方下载页面 `http://www.erlang.org/downloads` 获取 exe 安装包，直接打开并完成安装。
2. 安装 Rabbit MQ，通过官方下载页面 `https://www.rabbitmq.com/download.html` 获取 exe 安装包。
3. 下载完成后，直接运行安装程序。
4. Rabbit MQ Server 安装完成之后，会自动注册为服务，并以默认配置进行启动。



在 Windows 的安装过程中，有时候会碰到服务启动失败的情况，通常都是由于用户名为中文，导致默认的 db 和 log 目录访问出现问题。要解决该问题，需要先卸载 Rabbit MQ Server，然后设置环境变量 `RABBITMQ_BASE` 为一个不含中文的路径，比如 `E:\server\rabbitmq`。最后，重新安装 Rabbit MQ 即可。

在 Mac OS X 中安装

在 Mac OS X 中使用 brew 工具，可以很容易地安装 Rabbit MQ 的服务端，只需按如下命令操作即可：

1. 将 brew 更新到最新版本，执行 `brew update` 命令。
2. 安装 Erlang，执行 `brew install erlang` 命令。
3. 安装 Rabbit MQ Server，执行 `brew install rabbitmq` 命令。

通过执行上面的命令，Rabbit MQ Server 会被安装到 `/usr/local/sbin`，并不会自动加到用户的环境变量中去，所以我们需要在 `.bash_profile` 或 `.profile` 文件中增加下面的内容：

```
PATH=$PATH:/usr/local/sbin
```

这样，就可以通过 `rabbitmq-server` 命令来启动 Rabbit MQ 的服务端了。

在 Ubuntu 中安装

在 Ubuntu 中，我们可以使用 APT 仓库来进行安装：

1. 安装 Erlang，执行 `apt-get install erlang` 命令。
2. 执行下面的命令，新增 APT 仓库到 `/etc/apt/sources.list.d`：

```
echo 'deb http://www.rabbitmq.com/debian/ testing main' |
   sudo tee /etc/apt/sources.list.d/rabbitmq.list
```

3. 更新 APT 仓库的 `package list`，执行 `sudo apt-get update` 命令。

4. 安装 Rabbit Server，执行 `sudo apt-get install rabbitmq-server` 命令。

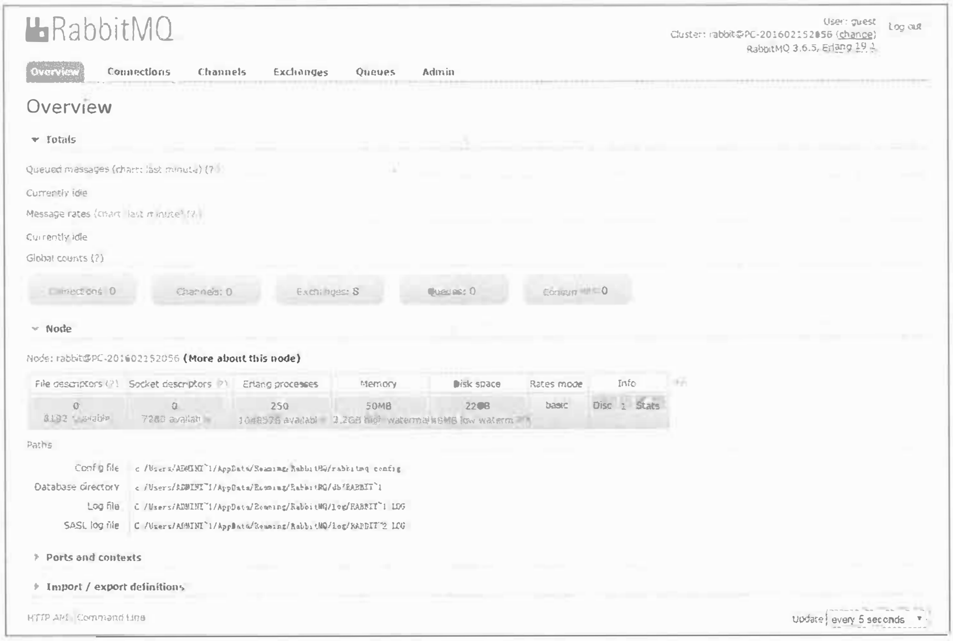
Rabbit 管理

我们可以直接通过访问配置文件进行管理，也可以通过访问 Web 进行管理。下面将介绍如何通过 Web 进行管理。

- 执行 `rabbitmq-plugins enable rabbitmq_management` 命令，开启 Web 管理插件，这样就可以通过浏览器来进行管理了。

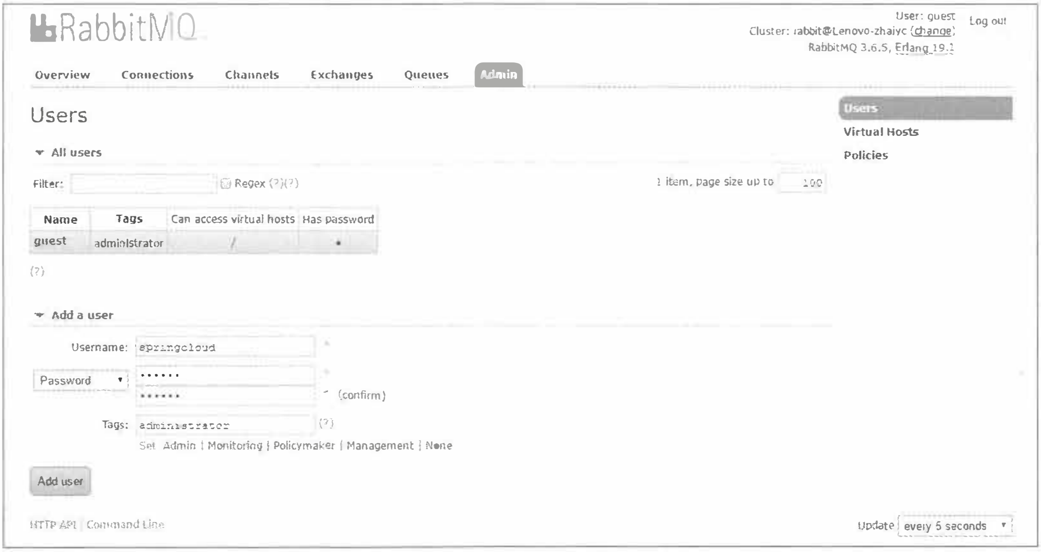
```
> rabbitmq-plugins enable rabbitmq_management
The following plugins have been enabled:
  mochiweb
  webmachine
  rabbitmq_web_dispatch
  amqp_client
  rabbitmq_management_agent
  rabbitmq_management

Applying plugin configuration to rabbit@PC-201602152056... started 6 plugins.
```

- 打开浏览器并访问 `http://localhost:15672/`，并使用默认用户 `guest` 登录，密码也为 `guest`。可以看到如下图所示的管理页面：



从图中我们可以看到之前章节中提到的一些基本概念，比如 Connections、Channels、Exchanges、Queues 等。第一次使用的读者，可以点开各项看看都有些什么内容，熟悉一下 RabbitMQ Server 的服务端。

- 单击 Admin 选项卡，如下图所示，可以尝试创建一个名为 `springcloud` 的用户。



其中，Tags 标签是 RabbitMQ 中的角色分类，共有下面几种。

- none：不能访问 management plugin。
- management：用户可以通过 AMQP 做的任何事外加如下内容。
  - 列出自己可以通过 AMQP 登入的 virtual hosts。
  - 查看自己的 virtual hosts 中的 queues、exchanges 和 bindings。
  - 查看和关闭自己的 channels 和 connections。
  - 查看有关自己的 virtual hosts 的“全局”统计信息，包含其他用户在这些 virtual hosts 中的活动。
- policymaker：management 可以做的任何事外加如下内容。
  - 查看、创建和删除自己的 virtual hosts 所属的 policies 和 parameters。
- monitoring：management 可以做的任何事外加如下内容。
  - 列出所有 virtual hosts，包括它们不能登录的 virtual hosts。
  - 查看其他用户的 connections 和 channels。
  - 查看节点级别的数据，如 clustering 和 memory 的使用情况。

- 查看真正的关于所有 virtual hosts 的全局的统计信息。

- administrator：policymaker 和 monitoring 可以做的任何事外加如下内容。
  - 创建和删除 virtual hosts。
  - 查看、创建和删除 users。
  - 查看、创建和删除 permissions。
  - 关闭其他用户的 connections。

快速入门

接下来，我们通过在 Spring Boot 应用中整合 RabbitMQ，实现一个简单的发送、接收消息的例子来对 RabbitMQ 有一个直观的感受和理解。

在 Spring Boot 中整合 RabbitMQ 是一件非常容易的事，因为之前我们已经介绍过 Starter POMs，其中的 AMQP 模块就可以很好地支持 RabbitMQ，下面我们就来详细说说整合过程。

- 新建一个 Spring Boot 工程，命名为 `rabbitmq-hello`。
- 在 `pom.xml` 中引入如下依赖内容，其中 `spring-boot-starter-amqp` 用于支持 RabbitMQ。

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-amqp</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

- 在 `application.properties` 中配置关于 RabbitMQ 的连接和用户信息，这里我们使用之前安装时创建的 `springcloud`。若没有自己的用户，可以回到上面的安装内容，在管理页面中创建用户。

```
spring.application.name=rabbitmq-hello

spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=springcloud
spring.rabbitmq.password=123456
```

- 创建消息生产者 `Sender`。通过注入 `AmqpTemplate` 接口的实例来实现消息的发送，`AmqpTemplate` 接口定义了一套针对 AMQP 协议的基础操作。在 Spring Boot 中会根据配置来注入其具体实现。在该生产者中，我们会产生一个字符串，并发送到名为 `hello` 的队列中。

```
@Component
public class Sender {

    @Autowired
    private AmqpTemplate rabbitTemplate;

    public void send() {
        String context = "hello " + new Date();
        System.out.println("Sender : " + context);
        this.rabbitTemplate.convertAndSend("hello", context);
    }

}
```

- 创建消息消费者 `Receiver`。通过`@RabbitListener` 注解定义该类对 `hello` 队列的监听，并用`@RabbitHandler` 注解来指定对消息的处理方法。所以，该消费者实现了对 `hello` 队列的消费，消费操作为输出消息的字符串内容。

```
@Component
@RabbitListener(queues = "hello")
public class Receiver {

    @RabbitHandler
    public void process(String hello) {
        System.out.println("Receiver : " + hello);
    }

}
```

- 创建 RabbitMQ 的配置类 `RabbitConfig`，用来配置队列、交换器、路由等高级信息。这里我们以入门为主，先以最小化的配置来定义，以完成一个基本的生产和消费过程。

```
@Configuration
public class RabbitConfig {
```

```
    @Bean
    public Queue helloQueue() {
        return new Queue("hello");
    }

}
```

- 创建应用主类。

```
@SpringBootApplication
public class HelloApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloApplication.class, args);
    }

}
```

- 创建单元测试类，用来调用消息生产。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = HelloApplication.class)
public class HelloApplicationTests {

    @Autowired
    private Sender sender;

    @Test
    public void hello() throws Exception {
        sender.send();
    }

}
```

完成程序编写之后，下面开始尝试运行。首先确保 RabbitMQ Server 已经启动，然后进行下面的操作。

- 启动应用主类，从控制台中，我们可看到如下内容，程序创建了一个访问 `127.0.0.1:5672` 中 `springcloud` 的连接。

```
o.s.a.r.c.CachingConnectionFactory : Created new connection: SimpleConnection@29836d32
[delegate=amqp://springcloud@127.0.0.1:5672/]
```

同时，我们通过 RabbitMQ 的控制面板，可以看到 Connections 和 Channels 中包含当前连接的条目。

- 运行单元测试类，我们可以在控制台中看到下面的输出内容，消息被发送到了 RabbitMQ Server 的 `hello` 队列中。

```
Sender : hello Sun Sep 25 11:06:11 CST 2016
```

- 切换到应用主类的控制台，我们可以看到类似如下的输出，消费者对 `hello` 队列的监听程序执行了，并输出了接收到的消息信息。

```
Receiver : hello Sun Sep 25 11:06:11 CST 2016
```

通过上面的示例，我们在 Spring Boot 应用中引入 `spring-boot-starter-amqp` 模块，进行简单配置就完成了对 RabbitMQ 的消息生产和消费的开发内容。然而在实际应用中，还有很多内容没有演示，比如之前提到的一些概念：交换机、路由关键字、绑定、虚拟主机等，这里不做更多的讲解，读者可以自行查阅 RabbitMQ 的官方教程，其中有更全面的讲解。在这里，我们需要重点理解的是，在整个生产消费过程中，生产和消费是一个异步操作，这也是在分布式系统中要使用消息代理的重要原因，以此我们可以使用通信来解耦业务逻辑。在这个例子中，读者可以进一步做一些测试，比如，不运行消费者，先运

行生产者，此时可以看到在 RabbitMQ Server 管理页面的 Queues 选项卡下多了一些待处理的消息，这时我们再启动消费者，它就会处理这些消息，所以通过生产消费模式的异步操作，系统间调用就没有同步调用需要那么高的实时性要求，同时也更容易控制处理的吞吐量以保证系统的正常运行等。

在上一节中，我们已经介绍了关于消息代理、AMQP 以及 RabbitMQ 的基础知识和使用方法。在下面的内容中，我们开始具体介绍 Spring Cloud Bus 的配置，并以一个 Spring Cloud Bus 与 Spring Cloud Config结合的例子来实现配置内容的实时更新。

先回顾一下，在上一章 Spring Cloud Config 的介绍中，我们留了一个悬念：如何实现对配置信息的实时更新。虽然我们已经能够通过`/refresh` 接口和 Git 仓库的 Web Hook 来实现 Git 仓库中的内容修改触发应用程序的属性更新。但是，若所有触发操作均需要我们手工去维护 Web Hook 中的应用配置的话，随着系统的不断扩展，会变得越来越难以维护，而消息代理中间件是解决该问题最为合适的方案。是否还记得我们在介绍消息代理中的特点时提到过这样一个功能：消息代理中间件可以将消息路由到一个或多个目的地。利用这个功能，我们就能完美地解决该问题，下面来说说 Spring Cloud Bus 中的具体实现方案。

整合 Spring Cloud Bus

因为 Spring Cloud 基于 Spring Boot，在上一节中我们已经体验了 Spring Boot 与 RabbitMQ 的整合，所以在 Spring Cloud Bus 中使用 RabbitMQ 也是非常容易配置的。

下面我们来具体动手尝试整个配置过程。

- 准备工作：这里我们不创建新的应用，但需要用到上一章中已经实现的关于 Spring Cloud Config的几个工程，若读者对其还不了解，建议先阅读第 8 章的内容。
  - `config-repo`：定义在Git仓库中的一个目录，其中存储了应用名为`didispace`的多环境配置文件，配置文件中有一个 `from` 参数。
  - `config-server-eureka`：配置了 Git 仓库，并注册到了 Eureka 的服务端。
  - `config-client-eureka`：通过 Eureka 发现 Config Server 的客户端，应用名为 `didispace`，用来访问配置服务器以获取配置信息。该应用中提供了一个`/from` 接口，它会获取 `config-repo/didispace-dev.properties` 中的 `from` 属性并返回。

- 扩展 config-client-eureka 应用。
  - 修改pom.xml，增加spring-cloud-starter-bus-amqp模块(注意spring-boot-starter-actuator 模块也是必需的，用来提供刷新端点)。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

- 在配置文件中增加关于 RabbitMQ 的连接和用户信息。

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=springcloud
spring.rabbitmq.password=123456
```

- 启动 config-server-eureka，再启动两个 config-client-eureka（分别在不同的端口上，比如 7002、7003）。我们可以在 config-client-eureka 中的控制台中看到如下内容，在启动时，客户端程序多了一个/bus/refresh 请求。

```
    o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped
"{[/bus/refresh],methods=[POST]}" onto public void
org.springframework.cloud.bus.endpoint.RefreshBusEndpoint.refresh(java.lang.String)
```

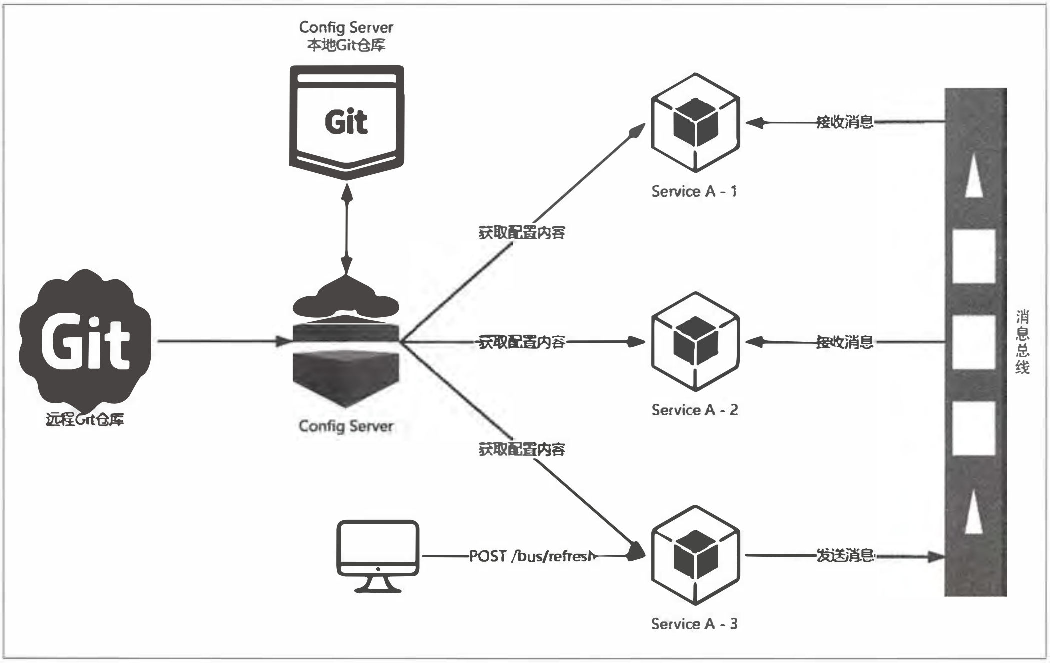
- 先访问两个 config-client-eureka 的/from 请求，会返回当前 config-repo/didispace-dev.properties 中的 from 属性。
- 接着，修改 config-repo/didispace-dev.properties 中的 from 属性值，并发送 PO ST 请求到其中的一个/bus/refresh。
- 最后，再分别访问启动的两个 config-client-eureka 的/from 请求，此时这两个请求都会返回最新的 config-repo/didispace-dev.properties 中的 from 属性。

到这里，我们已经能够通过 Spring Cloud Bus 来实时更新总线上的属性配置了。

原理分析

上一节中，我们通过使用 Spring Cloud Bus 与 Spring Cloud Config 的整合，并以 RabbitMQ 作为消息代理，实现了应用配置的动态更新。

整个方案的架构如下图所示，其中包含了 Git 仓库、Config Server 以及几个微服务应用的实例，这些微服务应用的实例中都引入了 Spring Cloud Bus，所以它们都连接到了 RabbitMQ 的消息总线上。



当我们将系统启动起来之后，图中“Service A”的三个实例会请求 Config Server 以获取配置信息，Config Server 根据应用配置的规则从 Git 仓库中获取配置信息并返回。

此时，若我们需要修改“Service A”的属性。首先，通过 Git 管理工具去仓库中修改对应的属性值，但是这个修改并不会触发“Service A”实例的属性更新。我们向“Service A”的实例 3 发送 POST 请求，访问`/bus/refresh`接口。此时，“Service A”的实例 3 就会将刷新请求发送到消息总线中，该消息事件会被“Service A”的实例 1 和实例 2 从总线中获取到，并重新从 Config Server 中获取它们的配置信息，从而实现配置信息的动态更新。

而从 Git 仓库中配置的修改到发起`/bus/refresh`的 POST 请求这一步可以通过 Git 仓库的 Web Hook 来自动触发。由于所有连接到消息总线上的应用都会接收到更新请求，所以在 Web Hook 中就不需要维护所有节点内容来进行更新，从而解决了上一章中仅通过 Web Hook 来逐个进行刷新的问题。

指定刷新范围

在上面的例子中，我们通过向服务实例请求 Spring Cloud Bus 的`/bus/refresh`接口，从而触发总线上其他服务实例的`/refresh`。但是在一些特殊场景下，我们希望可以刷新微服务中某个具体实例的配置。

Spring Cloud Bus 对这种场景也有很好的支持，/bus/refresh 接口提供了一个 destination 参数，用来定位具体要刷新的应用程序。比如，我们可以请求 /bus/refresh?destination=customers:9000，此时总线上的各应用实例会根据 destination 属性的值来判断是否为自己的实例名，若符合才进行配置刷新，若不符合就忽略该消息。

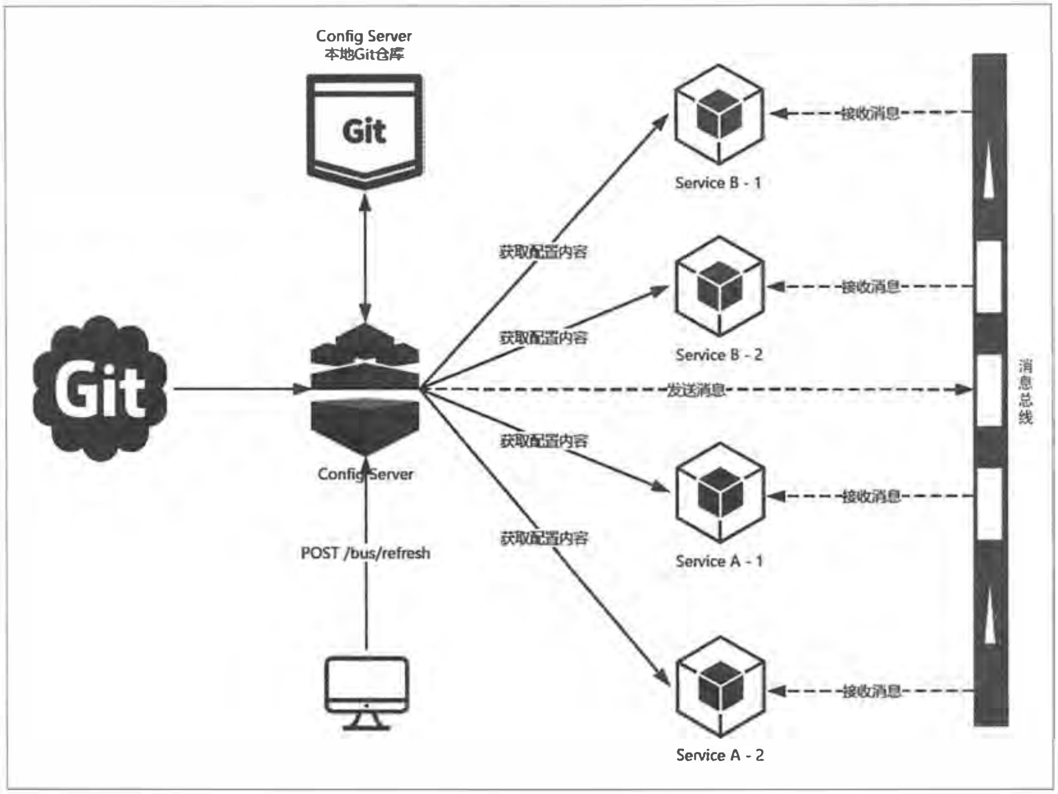
关于应用的实例名，我们在之前介绍 Spring Cloud Netflix 的 Eureka 时有过详细的介绍，它的默认命名按此规则生成：${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance_id:${server.port}}}，想了解更多内容可查看第 3 章的相关介绍。

destination 参数除了可以定位具体的实例之外，还可以用来定位具体的服务。定位服务的原理是通过使用 Spring 的 PathMatecher（路径匹配）来实现的，比如 /bus/refresh?destination=customers:**，该请求会触发 customers 服务的所有实例进行刷新。

架构优化

既然 Spring Cloud Bus 的/bus/refresh 接口提供了针对服务和实例进行配置更新的参数，那么我们的架构也可以相应做出一些调整。在之前的架构中，服务的配置更新需要通过向具体服务中的某个实例发送请求，再触发对整个服务集群的配置更新。虽然能实现功能，但是这样的结果是，我们指定的应用实例会不同于集群中的其他应用实例，这样会增加集群内部的复杂度，不利于将来的运维工作。比如，需要对服务实例进行迁移，那么我们不得不修改 Web Hook 中的配置等。所以要尽可能地让服务集群中的各个节点是对等的。

因此，我们将之前的架构做了一些调整，如下图所示。



我们主要做了以下这些改动：

1. 在 Config Server 中也引入 Spring Cloud Bus，将配置服务端也加入到消息总线中来。
2. `/bus/refresh` 请求不再发送到具体服务实例上，而是发送给 Config Server，并通过 `destination` 参数来指定需要更新配置的服务或实例。

通过上面的改动，我们的服务实例不需要再承担触发配置更新的职责。同时，对于 Git 的触发等配置都只需要针对 Config Server 即可，从而简化了集群上的一些维护工作。

RabbitMQ 配置

Spring Cloud Bus 中的 RabbitMQ 整合使用了 Spring Boot 的 `ConnectionFactory`，所以在 Spring Cloud Bus 中支持使用以 `spring.rabbit.mq` 为前缀的 Spring Boot 配置属性，具体的配置属性、说明以及默认值如下表所示。

属性名	说明	默认值
spring.rabbitmq.addresses	客户端连接的地址，有多个的时候使用逗号分隔，该地址可以是 IP 与 Port 的结合	
spring.rabbitmq.cache.channel.checkout-timeout	当缓存已满时，获取 Channel 的等待时间，单位为毫秒	
spring.rabbitmq.cache.channel.size	缓存中保持的 Channel 数量	
spring.rabbitmq.cache.connection.mode	连接缓存的模式	CHANNEL
spring.rabbitmq.cache.connection.size	缓存的连接数	
spring.rabbitmq.connection-timeout	连接超时参数，单位为毫秒；设置为“0”代表无穷大	
spring.rabbitmq.dynamic	默认创建一个 AmqpAdmin 的 Bean	true
spring.rabbitmq.host	RabbitMQ 的主机地址	localhost
spring.rabbitmq.listener.acknowledge-mode	容器的 acknowledge 模式	
spring.rabbitmq.listener.auto-startup	启动时自动启动容器	true
spring.rabbitmq.listener.concurrency	消费者的最小数量	
spring.rabbitmq.listener.default-requeue-rejected	投递失败时是否重新排队	true
spring.rabbitmq.listener.max-concurrency	消费者的最大数量	
spring.rabbitmq.listener.prefetch	在单个请求中处理的消息个数，它应该大于等于事务数量	
spring.rabbitmq.listener.retry.enabled	不论是不是重试的发布	false
spring.rabbitmq.listener.retry.initial-interval	第一次与第二次投递尝试的时间间隔	1000
spring.rabbitmq.listener.retry.max-attempts	尝试投递消息的最大数量	3
spring.rabbitmq.listener.retry.max-interval	两次尝试的最大时间间隔	10000
spring.rabbitmq.listener.retry.multiplier	上一次尝试时间间隔的乘数	1.0
spring.rabbitmq.listener.retry.stateless	不论重试是有状态的还是无状态的	true
spring.rabbitmq.listener.transaction-size	在一个事务中处理的消息数量。为了获得最佳效果，该值应设置为小于等于每个请求中处理的消息个数，即 `spring.rabbitmq.listener.prefetch` 的值	
spring.rabbitmq.password	登录到 RabbitMQ 的密码	
spring.rabbitmq.port	RabbitMQ 的端口号	5672
spring.rabbitmq.publisher-confirms	开启 Publisher Confirm 机制	false
spring.rabbitmq.publisher-returns	开启 Publisher Return 机制	false
spring.rabbitmq.requested-heartbeat	请求心跳超时时间，单位为秒	
spring.rabbitmq.ssl.enabled	启用 SSL 支持	false

续表

属性名	说明	默认值
spring.rabbitmq.ssl.key-store	保存 SSL 证书的地址	
spring.rabbitmq.ssl.key-store-password	访问 SSL 证书的地址使用的密码	
spring.rabbitmq.ssl.trust-store	SSL 的可信地址	
spring.rabbitmq.ssl.trust-store-password	访问 SSL 的可信地址的密码	
spring.rabbitmq.ssl.algorithm	SSL 算法，默认使用 Rabbit 的客户端算法库	
spring.rabbitmq.template.mandatory	启用强制消息	false
spring.rabbitmq.template.receive-timeout	`receive()` 方法的超时时间	0
spring.rabbitmq.template.reply-timeout	`sendAndReceive()` 方法的超时时间	5000
spring.rabbitmq.template.retry.enabled	设置为 true 的时候 `RabbitTemplate` 能够实现重试	false
spring.rabbitmq.template.retry.initial-interval	第一次与第二次发布消息的时间间隔	1000
spring.rabbitmq.template.retry.max-attempts	尝试发布消息的最大数量	3
spring.rabbitmq.template.retry.max-interval	尝试发布消息的最大时间间隔	10000
spring.rabbitmq.template.retry.multiplier	上一次尝试时间间隔的乘数	1.0
spring.rabbitmq.username	登录到 RabbitMQ 的用户名	
spring.rabbitmq.virtual-host	连接到 RabbitMQ 的虚拟主机	

Kafka 实现消息总线

Spring Cloud Bus 除了支持 RabbitMQ 的自动化配置之外，还支持现在被广泛应用的 Kafka。在本节中，我们将搭建一个 Kafka 的本地环境，并通过它来尝试使用 Spring Cloud Bus 对 Kafka 的支持，实现消息总线的功能。

Kafka 简介

Kafka 是一个由 LinkedIn 开发的分布式消息系统，它于 2011 年年初开源，现在由著名的 Apache 基金会维护与开发。Kafka 使用 Scala 实现，被用作 LinkedIn 的活动流和运营数据处理的管道，现在也被诸多互联网企业广泛地用作数据流管道和消息系统。

Kafka 是基于消息发布-订阅模式实现的消息系统，其主要设计目标如下所述。

- 消息持久化：以时间复杂度为 $O(1)$的方式提供消息持久化能力，即使对 TB 级以上的数据也能保证常数时间复杂度的访问性能。
- 高吞吐：在廉价的商用机器上也能支持单机每秒 10 万条以上的吞吐量。

- 分布式：支持消息分区以及分布式消费，并保证分区内的消息顺序。
- 跨平台：支持不同技术平台的客户端（如 Java、PHP、Python 等）。
- 实时性：支持实时数据处理和离线数据处理。
- 伸缩性：支持水平扩展。

Kafka 中涉及的一些基本概念，如下所示。

- Broker：Kafka 集群包含一个或多个服务器，这些服务器被称为 Broker。
- Topic：逻辑上同 RabbitMQ 的 Queue 队列相似，每条发布到 Kafka 集群的消息都必须有一个 Topic。（物理上不同 Topic 的消息分开存储，逻辑上一个 Topic 的消息虽然保存于一个或多个 Broker 上，但用户只需指定消息的 Topic 即可生产或消费数据而不必关心数据存于何处。）
- Partition：Partition 是物理概念上的分区，为了提供系统吞吐率，在物理上每个 Topic 会分成一个或多个 Partition，每个 Partition 对应一个文件夹（存储对应分区的消息内容和索引文件）。
- Producer：消息生产者，负责生产消息并发送到 Kafka Broker。
- Consumer：消息消费者，向 Kafka Broker 读取消息并处理的客户端。
- Consumer Group：每个 Consumer 属于一个特定的组（可为每个 Consumer 指定属于一个组，若不指定则属于默认组），组可以用来实现一条消息被组内多个成员消费等功能。

快速入门

在对 Kafka 有了一些基本了解之后，下面我们来尝试搭建一个 Kafka 服务端，并体验一下基于 Kafka 的消息生产与消费。

环境安装

首先，我们需要从官网上下载安装介质。下载地址为 `http://kafka.apache.org/downloads.html`。本例中采用的版本为 Kafka-0.10.0.1。

在解压 Kafka 的安装包之后，可以看到其目录结构如下所示：

```
kafka
  +-bin
    +-windows
  +-config
  +-libs
  +-logs
  +-site-docs
```

由于 Kafka 的设计中依赖了 ZooKeeper，所以我们在 bin 和 config 目录中除了看到 Kafka 相关的内容之外，还有 ZooKeeper 相关的内容。其中 bin 目录中存放了 Kafka 和 ZooKeeper 的命令行工具，bin 根目录下存放的是适用于 Linux/UNIX 的 shell，而 bin/windows 下存放的则是适用于 Windows 下的 bat。我们可以根据实际的系统来设置环境变量，以方便后续的使用和操作。而 config 目录，则用来存放关于 Kafka 与 ZooKeeper 的配置信息。

启动测试

下面我们来尝试启动 ZooKeeper 和 Kafka 来进行消息的生产和消费。示例中所有的命令均以配置了 Kafka 的环境变量为例。

- 启动 ZooKeeper，执行命令 zookeeper-server-start config/zookeeper.properties，该命令需要指定 ZooKeeper 的配置文件位置才能正确启动，Kafka 的压缩包中包含了其默认配置，开发与测试环境基本不需要修改，所以这里不做详细介绍，对于线上的调优需求，请读者自行查看官方文档进行操作。

```
    [2016-09-28 08:05:34,849] INFO Reading configuration from: config\zookeeper.
properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
    [2016-09-28 08:05:34,850] INFO autopurge.snapRetainCount set to 3 (org.apache.
zookeeper.server.DatadirCleanupManager)
    [2016-09-28 08:05:34,851] INFO autopurge.purgeInterval set to 0 (org.apache.
zookeeper.server.DatadirCleanupManager)
    [2016-09-28 08:05:34,851] INFO Purge task is not scheduled. (org.apache.zookeeper.
server.DatadirCleanupManager)
    [2016-09-28 08:05:34,852] WARN Either no config or no quorum defined in config,
running  in standalone mode (org.apache.zookeeper.server.quorum.QuorumPeerMain)
    [2016-09-28 08:05:34,868] INFO Reading configuration from: config\zookeeper.
properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
    [2016-09-28 08:05:34,869] INFO Starting server (org.apache.zookeeper.server.
ZooKeeperServerMain)
    ...
    [2016-09-28 08:05:34,940] INFO binding to port 0.0.0.0/0.0.0.0:2181 (org.apache.
zookeeper.server.NIOServerCnxnFactory)
```

从控制台信息中我们可以看到，ZooKeeper 从指定的 config/zookeeper.properties 配置文件中读取信息并绑定 2181 端口启动服务。有时候启动失败，可查看一下端口是否被占用，可以杀掉占用进程或通过修改 config/zookeeper.properties 配置文件中的 clientPort 内容以绑定其他端口号来启动 ZooKeeper。

- 启动 Kafka，执行命令 kafka-server-start config/server.properties，该命令也需要指定 Kafka 配置文件的正确位置，如上命令中指向了解压目录包含的默认配置。若在测试时，使用外部集中环境的 ZooKeeper 的话，我们可以在该配置

文件中通过 zookeeper.connect 参数来设置 ZooKeeper 的地址和端口，它默认会连接本地 2181 端口的 ZooKeeper；如果需要设置多个 ZooKeeper 节点，可以为这个参数配置多个 ZooKeeper 地址，并用逗号分隔。比如 zookeeper.connect=127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002。除此之外，该配置文件中还提供了关于服务端连接、日志等配置参数，具体的线上配置可根据实际情况进行调整。

- 创建 Topic：执行命令 kafka-topics --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test。通过该命令，创建了一个名为 test 的 Topic，该 Topic 包含一个分区和一个 Replica。在创建完成后，可以使用 kafka-topics --list --zookeeper localhost:2181 命令来查看当前的 Topic。

另外，如果不使用 kafka-topics 命令来手工创建，直接使用下面的内容进行消息创建时也会自动创建 Topics。

- 创建消息生产者，执行命令 kafka-console-producer --broker-list localhost:9092 --topic test。kafka-console-producer 命令可以启动 Kafka 基于命令行的消息生产客户端，启动后可以直接在控制台中输入消息来发送，控制台中的每一行数据都会被视为一条消息来发送。我们可以尝试输入几行消息，由于此时并没有消费者，所以这些输入的消息都会被阻塞在名为 test 的 Topics 中，直到有消费者将其消费掉。
- 创建消息消费者，执行命令 kafka-console-consumer --zookeeper localhost:2181 --topic test --from-beginning。kafka-console-consumer 命令启动的是 Kafka 基于命令行的消息消费客户端，启动之后，马上可以在控制台中看到输出了之前我们在消息生产客户端中发送的消息。我们可以再次打开之前的消息生产客户端来发送消息，并观察消费者这边对消息的输出来体验 Kafka 对消息的基础处理。

整合 Spring Cloud Bus

在介绍 Kafka 之前，我们已经通过引入 spring-cloud-starter-bus-amqp 模块，完成了使用 RabbitMQ 来实现消息总线。若我们要使用 Kafka 来实现消息总线时，只需把 spring-cloud-starter-bus-amqp 替换成 spring-cloud-starter-bus-kafka 模块，在 pom.xml 的 dependency 节点中进行修改，具体如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-kafka</artifactId>
</dependency>
```

如果在启动 Kafka 时均采用了默认配置，那么我们不需要再做任何其他配置就能在本地实现从 RabbitMQ 到 Kafka 的切换。可以尝试把刚刚搭建的 ZooKeeper、Kafka 启动起来，并将修改为 spring-cloud-starter-bus-kafka 模块的 config-server 和 config-client 启动起来。

在 config-server 启动时，我们可以在控制台中看到如下输出：

```
    2016-09-28 22:11:29.627  INFO 15144 --- [           main]
o.s.c.s.b.k.KafkaMessageChannelBinder    : Using kafka topic for outbound:
springCloudBus
    2016-09-28 22:11:29.642  INFO 15144 --- [-localhost:2181]
org.I0Itec.zkclient.ZkEventThread        : Starting ZkClient event thread.
    ...
    016-09-28 22:11:30.290  INFO 15144 --- [           main]
o.s.i.kafka.support.ProducerFactoryBean  : Using producer properties =>
{bootstrap.servers=localhost:9092, linger.ms=0, acks=1, compression.type=none,
batch.size=16384}
    2016-09-28 22:11:30.298  INFO 15144 --- [           main]
o.a.k.clients.producer.ProducerConfig    : ProducerConfig values:
    ...
    2016-09-28 22:11:30.322  INFO 15144 --- [           main]
o.s.c.s.b.k.KafkaMessageChannelBinder$1  : Adding
{message-handler:outbound.springCloudBus} as a subscriber to the 'springCloudBusOutput'
channel
    2016-09-28 22:11:30.322  INFO 15144 --- [           main]
o.s.integration.channel.DirectChannel    : Channel
'config-server:7001.springCloudBusOutput' has 1 subscriber(s).
    2016-09-28 22:11:30.322  INFO 15144 --- [           main]
o.s.c.s.b.k.KafkaMessageChannelBinder$1  : started outbound.springCloudBus
    ...
    2016-09-28 22:11:31.465  INFO 15144 --- [           main]
s.i.k.i.KafkaMessageDrivenChannelAdapter : started
org.springframework.integration.kafka.inbound.KafkaMessageDrivenChannelAdapter@4178
cb34
    2016-09-28 22:11:31.467  INFO 15144 --- [           main]
o.s.c.s.b.k.KafkaMessageChannelBinder$7  : Adding
{message-handler:inbound.springCloudBus.anonymous.8b9e6c7b-6a50-48c5-b981-8282a0d5a
30b} as a subscriber to the 'bridge.springCloudBus' channel
    2016-09-28 22:11:31.467  INFO 15144 --- [           main]
o.s.c.s.b.k.KafkaMessageChannelBinder$7  : started
inbound.springCloudBus.anonymous.8b9e6c7b-6a50-48c5-b981-8282a0d5a30b
```

从控制台的输出内容我们可以看到，config-server 连接到了 Kafka 中，并使用了名为 springCloudBus 的 Topic。

此时，我们可以使用 kafka-topics --list --zookeeper localhost:2181 命令来查看当前 Kafka 中的 Topic。若已成功启动了 config-server 并配置正确，可以在 Kafka 中看到已经多了一个名为 springCloudBus 的 Topic。

我们再启动配置了 spring-cloud-starter-bus-kafka 模块的 config-client，可以看到控制台中输出了如下内容：

```
    2016-09-28 22:43:55.067  INFO 6136 --- [           main]
o.s.c.s.b.k.KafkaMessageChannelBinder    : Using kafka topic for outbound:
springCloudBus
    2016-09-28 22:43:55.078  INFO 6136 --- [-localhost:2181]
org.I0Itec.zkclient.ZkEventThread        : Starting ZkClient event thread.
    ...
    2016-09-28 22:50:38.584  INFO 828 --- [           main]
o.s.i.kafka.support.ProducerFactoryBean  : Using producer properties =>
{bootstrap.servers=localhost:9092, linger.ms=0, acks=1, compression.type=none,
batch.size=16384}
    2016-09-28 22:50:38.592  INFO 828 --- [           main]
o.a.k.clients.producer.ProducerConfig    : ProducerConfig values:
    ...
    2016-09-28 22:50:38.615  INFO 828 --- [           main]
o.s.c.s.b.k.KafkaMessageChannelBinder$1  : Adding
{message-handler:outbound.springCloudBus} as a subscriber to the 'springCloudBusOutput'
channel
    2016-09-28 22:50:38.616  INFO 828 --- [           main]
o.s.integration.channel.DirectChannel    : Channel
'didispace:7002.springCloudBusOutput' has 1 subscriber(s).
    2016-09-28 22:50:38.616  INFO 828 --- [           main]
o.s.c.s.b.k.KafkaMessageChannelBinder$1  : started outbound.springCloudBus
    ...
    2016-09-28 22:50:39.162  INFO 828 --- [           main]
s.i.k.i.KafkaMessageDrivenChannelAdapter : started
org.springframework.integration.kafka.inbound.KafkaMessageDrivenChannelAdapter@60cf
855e
    2016-09-28 22:50:39.162  INFO 828 --- [           main]
o.s.c.s.b.k.KafkaMessageChannelBinder$7  : Adding
{message-handler:inbound.springCloudBus.anonymous.f8fc9c0c-ccd3-46dd-9537-07198f4ee
216} as a subscriber to the 'bridge.springCloudBus' channel
    2016-09-28 22:50:39.163  INFO 828 --- [           main]
o.s.c.s.b.k.KafkaMessageChannelBinder$7  : started
inbound.springCloudBus.anonymous.f8fc9c0c-ccd3-46dd-9537-07198f4ee216
```

可以看到，config-client 启动时输出了类似的内容，它们都订阅了名为 springCloudBus 的 Topic。从这里我们也可以知道，在消息总线上的节点，从结构上来说，不论是 config-server 还是 config-client，它们都是对等的。

在启动了 config-server 和 config-client 之后，为了更明显地观察消息总线刷新配置的效果，我们可以在本地启动多个不同端口的 config-client。此时，我们的 config-server 以及多个 config-client 都已经连接到了由 Kafka 实现的消息总线上。我们可以先访问各个 config-client 上的/from 请求，查看它获取到的配置内容。然后，修改 Git 中对应的参数内容，再访问各个 config-client 上的/from 请求，可以

看到配置内容并没有改变。最后，我们向 config-server 发送 POST 请求：/bus/refresh，此时再去访问各个 config-client 上的/from 请求，就能获得最新的配置信息，各客户端上的配置都已经加载为最新的 Git 配置内容。

从 config-client 的控制台中，我们可以看到如下内容：

```
    2016-09-29 08:20:34.361  INFO 21256 --- [ kafka-binder-1]
o.s.cloud.bus.event.RefreshListener     : Received remote refresh request. Keys
refreshed [from]
```

RefreshListener 监听类记录了收到远程刷新请求，并刷新了 from 属性的日志，在下一节中，我们将根据消息内容与日志输出信息作为线索来探索 Spring Cloud Bus 的工作机制。

Kafka 配置

在上面的例子中，由于 Kafka、ZooKeeper 均运行于本地，在自动化配置的支持下，我们没有在测试程序中通过配置信息来指定 Kafka 和 ZooKeeper 的配置信息，就完成了本地消息总线的试验。但是在实际应用中，Kafka 和 ZooKeeper 一般都会独立部署，所以在应用中需要为 Kafka 和 ZooKeeper 配置一些连接信息等。Kafka 的整合与 RabbitMQ 不同，在 Spring Boot 1.3.7 中并没有直接提供 Starter 模块，而是采用了 Spring CloudStream 的 Kafka 模块，所以对于 Kafka 的配置均采用了 spring.cloud.stream.kafka 前缀，具体的配置内容我们可以参考第 10 章的“绑定器配置”一节中关于 Kafka 配置的内容。

深入理解

在整合 Kafka 实现了消息总线之后，我们不妨继续使用 Kafka 提供的控制台消费者来看看，当执行/bus/refresh 时，消息消费者都获得了什么。通过前文我们从控制台中获得的信息可以知道，Spring Cloud Bus 使用了名为 springCloudBus 的 Topic，所以我们可以使用命令 kafka-console-consumer --zookeeper localhost:2181 --topic springCloudBus，启动对 springCloudBus 的消费者控制台来进行观察。

启动消费者控制台之后，我们向 config-server 发送 POST 请求：/bus/refresh，此时在控制台中可以看到类似如下的内容：

```
contentType  "application/json"
{
    "type": "RefreshRemoteApplicationEvent",
    "timestamp": 1475073160814,
    "originService": "config-server:7001",
    "destinationService": "*:**",
    "id": "bbfbf495-39d8-4ff9-93d6-174873ff7299"
```

```
}

contentType  "application/json"
{
    "type": "AckRemoteApplicationEvent",
    "timestamp": 1475073160821,
    "originService": "config-server:7001",
    "destinationService": "*:**",
    "id": "1f794774-10d6-4140-a80d-470983c6c0ff",
    "ackId": "bbfbf495-39d8-4ff9-93d6-174873ff7299",
    "ackDestinationService": "*:**",
    "event": "org.springframework.cloud.bus.event.RefreshRemoteApplicationEvent"
}

contentType  "application/json"
{
    "type": "AckRemoteApplicationEvent",
    "timestamp": 1475075467554,
    "originService": "didispace:7002",
    "destinationService": "*:**",
    "id": "7560151e-f60c-49cd-8167-b691e846ad08",
    "ackId": "21502725-28f5-4d19-a98a-f8114fa4f1dc",
    "ackDestinationService": "*:**",
    "event": "org.springframework.cloud.bus.event.RefreshRemoteApplicationEvent"
}
...
```

下面，我们来详细理解消息中的信息内容。

- `type`：消息的事件类型。在上面的例子中，包含了 `RefreshRemoteApplicationEvent` 和 `AckRemoteApplicationEvent`。其中，`RefreshRemoteApplicationEvent` 事件就是我们用来刷新配置的事件，而 `AckRemoteApplicationEvent` 是响应消息已经正确接收的告知消息事件。
- `timestamp`：消息的时间戳。
- `originService`：消息的来源服务实例。
- `destinationService`：消息的目标服务实例。上面示例中的`*:**`代表了总线上的所有服务实例。如果想要指定服务或是实例，在之前介绍 RabbitMQ 实现消息总线时已经提过，只需要通过使用 `destination` 参数来定位具体要刷新的应用实例即可，比如发起`/bus/refresh?destination=didispace` 请求，就可以得到如下的刷新事件消息，其中 `destinationService` 为 `didispace:**`，表示总线上所有 `didispace` 服务的实例。

```
contentType  "application/json"
{
    "type": "RefreshRemoteApplicationEvent",
    "timestamp": 1475131215007,
    "originService": "config-server:7001",
    "destinationService": "didispace:**",
    "id": "667fe948-e9b2-447f-be22-3c8acf647ead"
}
```

- id：消息的唯一标识。

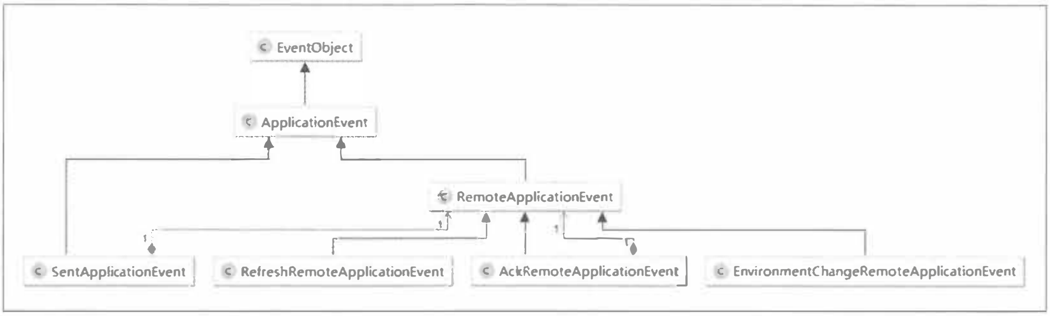
上面的消息内容是 RefreshRemoteApplicationEvent 和 AckRemoteApplicationEvent 类型共有的，下面几个属性是 AckRemoteApplicationEvent 所特有的，分别表示如下含义。

- ackId：Ack 消息对应的消息来源。我们可以看到第一条 AckRemoteApplicationEvent 的 ackId 对应了 RefreshRemoteApplicationEvent 的 id，说明这条 Ack 是告知该 RefreshRemoteApplicationEvent 事件的消息已经被收到。
- ackDestinationService：Ack 消息的目标服务实例。可以看到这里使用的是 *:**，所以消息总线上所有的实例都会收到该 Ack 消息。
- event：Ack 消息的来源事件。可以看到上例中的两个 Ack 均来源于刷新配置的 RefreshRemoteApplicationEvent 事件，我们在测试的时候由于启动了两个 config-client，所以有两个实例接收到了配置刷新事件，同时它们都会返回一个 Ack 消息。由于 ackDestinationService 为*:**，所以两个 config-client 都会收到对 RefreshRemoteApplicationEvent 事件的 Ack 消息。

源码分析

通过上面的分析，我们已经得到了两个非常重要的线索 RefreshRemoteApplicationEvent 和 AckRemoteApplicationEvent。我们不妨顺着这两个事件类来详细看看 Spring Cloud Bus 的源码，以帮助我们理解它的运行机制。

顺着 RefreshRemoteApplicationEvent 和 AckRemoteApplicationEvent，我们可以整理出如下的事件关系类图。

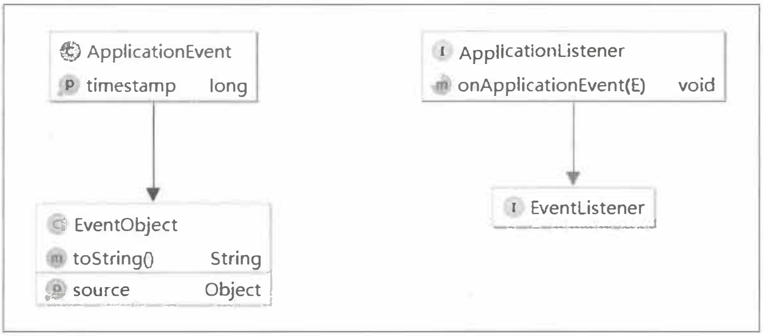


可以看到，其中 RefreshRemoteApplicationEvent 和 AckRemoteApplication-Event 这些我们已经接触过的事件都继承了 RemoteApplicationEvent 抽象类，而 RemoteApplicationEvent 继承自 Spring Framework 的 ApplicationEvent，可以断定，Spring Cloud Bus 也采用了 Spring 的事件驱动模型。

事件驱动模型

如果读者对 Spring 的事件驱动模型已经非常了解，那么可以跳过这一小节，直接看后面的分析。如果你还不清楚它的原理，建议先通过本小节的内容来理解其基本原理，以帮助阅读和理解后续的源码分析内容。

Spring 的事件驱动模型中包含了三个基本概念：事件、事件监听者和事件发布者，如下图所示。



- 事件：Spring 中定义了事件的抽象类 ApplicationEvent，它继承自 JDK 的 EventObject 类。从图中我们可以看到，事件包含了两个成员变量：timestamp，该字段用于存储事件发生的时间戳，以及父类中的 source，该字段表示源事件对象。当我们需要自定义事件的时候，只需要继承 ApplicationEvent，比如 RemoteApplicationEvent、RefreshRemoteApplicationEvent 等，可以在自定义的 Event 中增加一些事件的属性来给事件监听者处理。

- **事件监听者**：Spring 中定义了事件监听者的接口 ApplicationListener，它继承自 JDK 的 EventListener 接口，同时 ApplicationListener 接口限定了 ApplicationEvent 子类作为该接口中 onApplicationEvent(E event);函数的参数。所以，每一个 ApplicationListener 都是针对某个 ApplicationEvent 子类的监听和处理者。

那么，事件与监听者是如何关联起来的呢？我们看下图：

- **事件发布者**：Spring 中定义了 ApplicationEventPublisher 和 ApplicationEventMulticaster 两个接口用来发布事件。其中 ApplicationEventPublisher 接口定义了发布事件的函数 publishEvent(ApplicationEvent event) 和 publishEvent(Object event)；而 ApplicationEventMulticaster 接口中定义了对 ApplicationListener 的维护操作（比如新增、移除等）以及将 ApplicationEvent 多播给可用 ApplicationListener 的操作。

ApplicationEventPublisher 的 publishEvent 实现在 AbstractApplicationContext 中，具体如下：

```
protected void publishEvent(Object event, ResolvableType eventType) {
    Assert.notNull(event, "Event must not be null");
    ...
    if (this.earlyApplicationEvents != null) {
      this.earlyApplicationEvents.add(applicationEvent);
    }
    else {
      getApplicationEventMulticaster().multicastEvent(applicationEvent,
eventType);
    }
    ...
}
```

可以看到，它最终会调用 ApplicationEventMulticaster 的 multicastEvent 来具体实现发布事件给监听者的操作。而 ApplicationEventMulticaster 在 Spring 的默认实现位于 SimpleApplicationEventMulticaster 中，具体如下：

```
    public void multicastEvent(final ApplicationEvent event, ResolvableType eventType) {
      ResolvableType type = (eventType != null ? eventType :
resolveDefaultEventType(event));
      for (final ApplicationListener<?> listener : getApplicationListeners(event, type)) {
        Executor executor = getTaskExecutor();
        if (executor != null) {
          executor.execute(new Runnable() {
            @Override
            public void run() {
              invokeListener(listener, event);
            }
          });
        }
        else {
          invokeListener(listener, event);
        }
      }
    }

    protected void invokeListener(ApplicationListener listener, ApplicationEvent event) {
      ErrorHandler errorHandler = getErrorHandler();
      if (errorHandler != null) {
        try {
          listener.onApplicationEvent(event);
        }
        catch (Throwable err) {
          errorHandler.handleError(err);
        }
      }
      else {
        try {
          listener.onApplicationEvent(event);
        }
        catch (ClassCastException ex) {
          LogFactory.getLog(getClass()).debug("Non-matching event type for listener: "
+ listener, ex);
        }
      }
    }
```

SimpleApplicationEventMulticaster通过遍历维护的ApplicationListener集合来找到对应ApplicationEvent 的监听器，然后调用监听器的 onApplicationEvent 函数来对具体事件做出处理操作。

事件定义

在对 Spring 的事件模型有了一定的理解之后，下面我们来详细介绍 Spring Cloud Bus 中的事件定义。首先，从 RemoteApplicationEvent 抽象类开始：

```
@SuppressWarnings("serial")
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "type")
@JsonIgnoreProperties("source")
public abstract class RemoteApplicationEvent extends ApplicationEvent {
    private static final Object TRANSIENT_SOURCE = new Object();
    private final String originService;
    private final String destinationService;
    private final String id;

    protected RemoteApplicationEvent() {
        // for serialization libs like jackson
        this(TRANSIENT_SOURCE, null, null);
    }

    protected RemoteApplicationEvent(Object source, String originService,
            String destinationService) {
        super(source);
        this.originService = originService;
        if (destinationService == null) {
            destinationService = "*";
        }
        if (!destinationService.contains(":")) {
            // All instances of the destination unless specifically requested
            destinationService = destinationService + ":**";
        }
        this.destinationService = destinationService;
        this.id = UUID.randomUUID().toString();
    }

    protected RemoteApplicationEvent(Object source, String originService) {
        this(source, originService, null);
    }

    ...
}
```

先来看看 RemoteApplicationEvent 类上修饰的注解。

- @JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "type")：Jackson 对多态类型的处理注解，当进行序列化时，会使用子类的名称作为 type 属性的值，比如之前示例中的"type": "RefreshRemoteApplicationEvent"。

- @JsonIgnoreProperties("source")：序列化的时候忽略 source 属性，source 是 ApplicationEvent 的父类 EventObject 的属性，用来定义事件的发生源。

再来看看它的属性：originService、destinationService、id，这些内容都可以在 RemoteApplicationEvent 的子类事件消息中找到，比如：

```
{
    "type": "RefreshRemoteApplicationEvent",
    "timestamp": 1475073160814,
    "originService": "config-server:7001",
    "destinationService": "*:**",
    "id": "bbfbf495-39d8-4ff9-93d6-174873ff7299"
}

{
    "type": "AckRemoteApplicationEvent",
    "timestamp": 1475075467554,
    "originService": "didispace:7002",
    "destinationService": "*:**",
    "id": "7560151e-f60c-49cd-8167-b691e846ad08",
    "ackId": "21502725-28f5-4d19-a98a-f8114fa4f1dc",
    "ackDestinationService": "*:**",
    "event": "org.springframework.cloud.bus.event.RefreshRemoteApplicationEvent"
}
```

下面，我们再来分别看看 RemoteApplicationEvent 的几个具体实现的事件类。

- RefreshRemoteApplicationEvent 事件类，该事件用于远程刷新应用的配置信息。它的实现非常简单，只是继承了 RemoteApplicationEvent，并没有增加其他内容。从之前的示例中我们也能看到，消息中的内容与 RemoteApplication-Event 中包含的属性完全一致。

```
@SuppressWarnings("serial")
public class RefreshRemoteApplicationEvent extends RemoteApplicationEvent {

    @SuppressWarnings("unused")
    private RefreshRemoteApplicationEvent() {
        // for serializers
    }

    public RefreshRemoteApplicationEvent(Object source, String originService,
            String destinationService) {
```

```
        super(source, originService, destinationService);
    }
}
```

- AckRemoteApplicationEvent 事件类，该事件用于告知某个事件消息已经被接收，通过该消息我们可以监控各个事件消息的响应。从其成员属性中，我们可以找到之前示例中所总结的，比 RefreshRemoteApplicationEvent 事件的消息多出的几个属性：ackId、ackDestinationService 以及 event。其中 event 成员变量通过泛型限定了必须为 RemoteApplicationEvent 的子类对象，该定义符合这样的逻辑：Ack 消息肯定有一个事件源头，而每一个事件都必须继承 RemoteApplicationEvent 抽象类，所以 AckRemoteApplicationEvent 的事件源头肯定是一个 RemoteApplicationEvent 的子类，比如示例中的 Ack 消息源头就是 RemoteApplicationEvent 的子类事件：RefreshRemoteApplicationEvent。

```
@SuppressWarnings("serial")
public class AckRemoteApplicationEvent extends RemoteApplicationEvent {

    private final String ackId;
    private final String ackDestinationService;
    private final Class<? extends RemoteApplicationEvent> event;

    @SuppressWarnings("unused")
    private AckRemoteApplicationEvent() {
        super();
        this.ackDestinationService = null;
        this.ackId = null;
        this.event = null;
    }

    public AckRemoteApplicationEvent(Object source, String originService,
            String destinationService, String ackDestinationService, String ackId,
            Class<? extends RemoteApplicationEvent> type) {
        super(source, originService, destinationService);
        this.ackDestinationService = ackDestinationService;
        this.ackId = ackId;
        this.event = type;
    }

    ...
}
```

- EnvironmentChangeRemoteApplicationEvent 事件类，该事件用于动态更新消息总线上每个节点的 Spring 环境属性。可以看到，该类中定义了一个 Map 类型

的成员变量，而接收消息的节点就是根据该 Map 对象中的属性来覆盖本地的 Spring 环境属性。

```
@SuppressWarnings("serial")
public class EnvironmentChangeRemoteApplicationEvent extends
RemoteApplicationEvent {

    private final Map<String, String> values;

    @SuppressWarnings("unused")
    private EnvironmentChangeRemoteApplicationEvent() {
        // for serializers
        values = null;
    }

    public EnvironmentChangeRemoteApplicationEvent(Object source, String originService,
            String destinationService, Map<String, String> values) {
        super(source, originService, destinationService);
        this.values = values;
    }
    ...
}
```

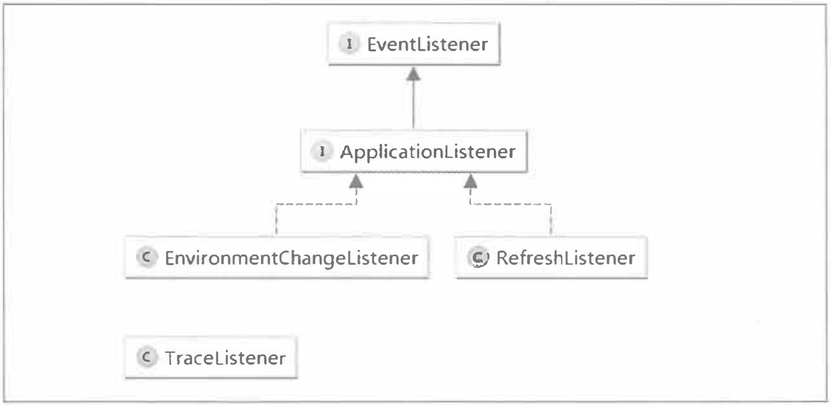
- `SentApplicationEvent` 事件类，细心的读者可能已经发现，该类的结构和内容与 `RemoteApplicationEvent` 非常相似，不同的是：该类不是抽象类，并且多一个成员 `Class<? extends RemoteApplicationEvent> type`。`SentApplicationEvent` 事件较为特殊，它主要用于发送信号来表示一个远程的事件已经在系统中被发送到某些地方了，从它的继承关系中，我们可以知道它本身并不是一个远程的事件（不是继承自 `RemoteApplicationEvent`），所以它不会被发送到消息总线上去，而是在本地产生（通常是由于响应了某个远程的事件）。由于该事件的 `id` 属性能够匹配消费者 `AckRemoteApplicationEvent` 消息中的 `ackId`，所以应用程序可以通过监听这个事件来监控远程事件消息的消费情况。

```
@SuppressWarnings("serial")
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "type")
@JsonIgnoreProperties("source")
public class SentApplicationEvent extends ApplicationEvent {

    private static final Object TRANSIENT_SOURCE = new Object();
    private final String originService;
    private final String destinationService;
    private final String id;
    private Class<? extends RemoteApplicationEvent> type;
```

```
    protected SentApplicationEvent() {
        // for serialization libs like jackson
        this(TRANSIENT_SOURCE, null, null, null, RemoteApplicationEvent.class);
    }

    public SentApplicationEvent(Object source, String originService,
            String destinationService, String id,
            Class<? extends RemoteApplicationEvent> type) {
        super(source);
        this.originService = originService;
        this.type = type;
        if (destinationService == null) {
            destinationService = "*";
        }
        if (!destinationService.contains(":")) {
            // All instances of the destination unless specifically requested
            destinationService = destinationService + ":**";
        }
        this.destinationService = destinationService;
        this.id = id;
    }

    ...
}
```

事件监听器

在了解了 Spring Cloud Bus 中的事件类之后，我们来看看另外一个重要元素：事件监听器。通过整理源码，可以得到下面的类图关系。



其中，`RefreshListener` 和 `EnvironmentChangeListener` 都继承了 Spring 事

件模型中的监听器接口 ApplicationListener。我们先来看看 RefreshListener：

```
public class RefreshListener
        implements ApplicationListener<RefreshRemoteApplicationEvent> {

    private static Log log = LogFactory.getLog(RefreshListener.class);

    private ContextRefresher contextRefresher;

    public RefreshListener(ContextRefresher contextRefresher) {
        this.contextRefresher = contextRefresher;
    }

    @Override
    public void onApplicationEvent(RefreshRemoteApplicationEvent event) {
        Set<String> keys = contextRefresher.refresh();
        log.info("Received remote refresh request. Keys refreshed " + keys);
    }
}
```

从泛型中我们可以看到该监听器就是针对我们之前所介绍的 RefreshRemote-ApplicationEvent 事件的，其中 onApplicationEvent 函数中调用了 ContextRefresher 中的 refresh()函数进行配置属性的刷新。

```
public class ContextRefresher {
    ...
    private ConfigurableApplicationContext context;
    ...
    public synchronized Set<String> refresh() {
        Map<String, Object> before = extract(
                this.context.getEnvironment().getPropertySources());
        addConfigFilesToEnvironment();
        Set<String> keys = changes(before,

extract(this.context.getEnvironment().getPropertySources())).keySet();
        this.context.publishEvent(new EnvironmentChangeEvent(keys));
        this.scope.refreshAll();
        return keys;
    }
    ...
}
```

再来看看 EnvironmentChangeListener 监听器。

```
public class EnvironmentChangeListener
        implements ApplicationListener<EnvironmentChangeRemoteApplicationEvent> {
```

```
    private static Log log = LogFactory.getLog(EnvironmentChangeListener.class);

    @Autowired
    private EnvironmentManager env;

    @Override
    public void onApplicationEvent(EnvironmentChangeRemoteApplicationEvent event) {
        Map<String, String> values = event.getValues();
        log.info("Received remote environment change request. Keys/values to update "
                + values);
        for (Map.Entry<String, String> entry : values.entrySet()) {
            env.setProperty(entry.getKey(), entry.getValue());
        }
    }
}
```

它是针对 EnvironmentChangeRemoteApplicationEvent 事件的监听类，在处理类中，可以看到它从 EnvironmentChangeRemoteApplicationEvent 中获取了之前提到的事件中定义的 Map 对象，然后通过遍历来更新 EnvironmentManager 中的属性内容。

事件跟踪

除了上面介绍的RefreshListener和EnvironmentChangeListener监听器外，还有一个与它们都有点不同的 TraceListener 监听器。

```
public class TraceListener {

    private static Log log = LogFactory.getLog(TraceListener.class);

    private TraceRepository repository;

    public TraceListener(TraceRepository repository) {
        this.repository = repository;
    }

    @EventListener
    public void onAck(AckRemoteApplicationEvent event) {
        this.repository.add(getReceivedTrace(event));
    }

    @EventListener
    public void onSend(SentApplicationEvent event) {
        this.repository.add(getSentTrace(event));
    }
```

```
    protected Map<String, Object> getSentTrace(SentApplicationEvent event) {
        ...
    }

    protected Map<String, Object> getReceivedTrace(AckRemoteApplicationEvent event) {
        ...
    }
}
```

从之前整理的类图和源码中，我们都可以看到该监听器并没有实现 ApplicationListener 接口，但可以看到这里使用了@EventListener 注解。该注解是从 Spring 4.2 开始提供的新功能，通过它可以自动地将函数注册为一个 ApplicationListener 的实现。所以在该类中，实际上等价于实现了两个监听器，一个监听 AckRemoteApplicationEvent 事件，一个监听 SentApplicationEvent 事件。

在这两个监听处理函数中调用了类似的方法：this.repository.add(getReceivedTrace(event));，其中 TraceRepository 是对 Trace 跟踪信息的操作接口，而它的默认实现是 spring-boot-actuator 模块的 InMemoryTraceRepository，具体如下：

```
public class InMemoryTraceRepository implements TraceRepository {

    private int capacity = 100;
    private boolean reverse = true;
    private final List<Trace> traces = new LinkedList<Trace>();

    public void setReverse(boolean reverse) {
        synchronized (this.traces) {
            this.reverse = reverse;
        }
    }

    public void setCapacity(int capacity) {
        synchronized (this.traces) {
            this.capacity = capacity;
        }
    }

    @Override
    public List<Trace> findAll() {
        synchronized (this.traces) {
            return Collections.unmodifiableList(new ArrayList<Trace>(this.traces));
        }
    }
```

```
        @Override
        public void add(Map<String, Object> map) {
            Trace trace = new Trace(new Date(), map);
            synchronized (this.traces) {
                while (this.traces.size() >= this.capacity) {
                    this.traces.remove(this.reverse ? this.capacity - 1 : 0);
                }
                if (this.reverse) {
                    this.traces.add(0, trace);
                }
                else {
                    this.traces.add(trace);
                }
            }
        }

    }
```

可以看到，默认的 Trace 跟踪信息存储并没有用到特别的数据库或消息系统，而是采用了内存存储的方式。如上代码所示，通过 `LinkedList<Trace>`集合和 `capacity` 属性的定义，在 `add(Map<String, Object> map)`函数中进行循环存储，所以默认的 Trace 跟踪实现只能存储和查询最近的 100 条跟踪信息。

那么跟踪事件都记录了哪些内容呢？我们继续看 `TraceListener` 中 `getSentTrace` 和 `getReceivedTrace` 的具体实现：

```
    public class TraceListener {
        ...
        protected Map<String, Object> getSentTrace(SentApplicationEvent event) {
            Map<String, Object> map = new LinkedHashMap<String, Object>();
            map.put("signal", "spring.cloud.bus.sent");
            map.put("type", event.getType().getSimpleName());
            map.put("id", event.getId());
            map.put("origin", event.getOriginService());
            map.put("destination", event.getDestinationService());
            if (log.isDebugEnabled()) {
                log.debug(map);
            }
            return map;
        }

        protected Map<String, Object> getReceivedTrace(AckRemoteApplicationEvent event) {
            Map<String, Object> map = new LinkedHashMap<String, Object>();
            map.put("signal", "spring.cloud.bus.ack");
            map.put("event", event.getEvent().getSimpleName());
```

```
        map.put("id", event.getAckId());
        map.put("origin", event.getOriginService());
        map.put("destination", event.getAckDestinationService());
        if (log.isDebugEnabled()) {
            log.debug(map);
        }
        return map;
    }
}
```

可以看到，这两个函数会收集关于发送和接收到的 Ack 事件信息，并且两个函数获得的内容就是事件定义相关的一些属性，看到这里大家是否感觉似曾相识？是的，这些信息与之前我们通过 Kafka 的控制台工具获取的消息内容非常类似。既然 Spring Cloud Bus 已经提供了 Trace 跟踪信息的监听和记录，我们不妨尝试使用一下。要开启该功能非常简单，只需在配置文件中将下面的属性设置为 true 即可：

```
spring.cloud.bus.trace.enabled=true
```

通过请求配置主机的`/trace`接口，比如`http://localhost:7002/trace`，可以获得如下信息，

```
[
  {
    "timestamp": 1475129670494,
    "info": {
      "signal": "spring.cloud.bus.ack",
      "event": "RefreshRemoteApplicationEvent",
      "id": "84ecdf83-a904-41bc-a34d-62680ccf35d7",
      "origin": "config-server:7001",
      "destination": "*:**"
    }
  },
  {
    "timestamp": 1475129670475,
    "info": {
      "signal": "spring.cloud.bus.sent",
      "type": "RefreshRemoteApplicationEvent",
      "id": "84ecdf83-a904-41bc-a34d-62680ccf35d7",
      "origin": "config-server:7001",
      "destination": "*:**"
    }
  },
  {
    "timestamp": 1475129670473,
    "info": {
      "signal": "spring.cloud.bus.ack",
```

```
        "event": "RefreshRemoteApplicationEvent",
        "id": "84ecdf83-a904-41bc-a34d-62680ccf35d7",
        "origin": "didispace:7002",
        "destination": "*:**"
      }
    }
]
```

与我们分析的内容一样，该请求返回了最近的 Send 和 Ack 消息内容。

如果希望针对 AckRemoteApplicationEvent 或是 SentApplicationEvent 做一些特殊处理，我们也可以通过@EventListener 注解在应用程序中编写自己的处理逻辑，或者重写 TraceRepository 来改造跟踪的存储等。

原则上每一个消息总线上的应用都可以用来跟踪 Ack 消息，但是大多数情况下我们把这个任务交给更核心的服务（比如特定的监控服务），这样在该服务中我们就能在 Ack 消息中实现更复杂的逻辑进行预警和善后工作。

事件发布

通过上面的分析，我们已经了解了 Spring Cloud Bus 中事件以及监听器的定义，下面来看看这些事件是如何发布给监听器进行处理的。

在 org.springframework.cloud.bus 包下，我们可以找到关于 Spring Cloud Bus 启动时加载的一些基础类和接口，包括自动化配置类 BusAutoConfiguration、属性定义类 BusProperties 等。我们可以从 Spring Cloud Bus 的自动化配置类中看看它在启动的时候都加载了什么内容：

```
@Configuration
@ConditionalOnBusEnabled
@EnableBinding(SpringCloudBusClient.class)
@EnableConfigurationProperties(BusProperties.class)
public class BusAutoConfiguration implements ApplicationEventPublisherAware {

    public static final String BUS_PATH_MATCHER_NAME = "busPathMatcher";

    @Autowired
    @Output(SpringCloudBusClient.OUTPUT)
    private MessageChannel cloudBusOutboundChannel;

    @Autowired
    private ServiceMatcher serviceMatcher;

    @Autowired
    private ChannelBindingServiceProperties bindings;
```

```
    @Autowired
    private BusProperties bus;

    private ApplicationEventPublisher applicationEventPublisher;
    ...
}
```

我们先来看看在该自动化配置类中，都定义了哪些成员。

- MessageChannel cloudBusOutboundChannel：该接口定义了发送消息的抽象方法。
- ServiceMatcher serviceMatcher：该对象中提供了下面两个重要函数，用来判断事件的来源服务是否为自己，以及判断目标是否为自己，以此作为依据是否要响应消息进行事件的处理。

```
public boolean isFromSelf(RemoteApplicationEvent event) {
    String originService = event.getOriginService();
    String serviceId = getServiceId();
    return this.matcher.match(originService, serviceId);
}

public boolean isForSelf(RemoteApplicationEvent event) {
    String destinationService = event.getDestinationService();
    return (destinationService == null || destinationService.trim().isEmpty() ||
this.matcher.match(destinationService, getServiceId()));
}
```

- ChannelBindingServiceProperties bindings：定义了消息服务的绑定属性。
- BusProperties bus：该对象定义了 Spring Cloud Bus 的属性，具体如下所示。

```
@ConfigurationProperties("spring.cloud.bus")
public class BusProperties {

    private Env env = new Env();
    private Refresh refresh = new Refresh();
    private Ack ack = new Ack();
    private Trace trace = new Trace();
    private String destination = "springCloudBus";
    private boolean enabled = true;
    ...
}
```

从中可以看到，Spring Cloud Bus 的属性前缀使用了 spring.cloud.bus。

destination 和 enabled 属性分别定义了默认的队列（Queue）或主题（Topic）是否连接到消息总线，所以我们可以通过 spring.cloud.bus.destination 来修改消息总线使用的队列或主题名称，以及使用 spring.cloud.bus.enabled 属性来设置应用是否要连接到消息总线上。

另外，在该配置类中为 Env、Refresh、Ack、Trace 4 种已经实现的事件分别创建了配置对象，这些配置类都是 BusProperties 的内部类。从下面的源码中，我们可以看到对于这 4 种事件，Env、Refresh、Ack 均是默认开启的，只有 Trace 事件需要通过修改配置来开启，就如之前我们介绍“事件跟踪”的时候配置 spring.cloud.bus.trace.enabled=true 属性那样。

```
public static class Env {

    private boolean enabled = true;
    ...
}

public static class Refresh {

    private boolean enabled = true;
    ...
}

public static class Ack {

    private boolean enabled = true;
    private String destinationService;
    ...
}

public static class Trace {

    private boolean enabled = false;
    ...
}
```

- ApplicationEventPublisher：Spring 事件模型中用来发布事件的接口，也就是我们之前介绍的事件以及监听的桥梁。

除了定义的这些成员变量之外，还能看到这里定义了两个监听方法 acceptLocal 和 acceptRemote。

其中，acceptLocal 方法如下所示，它通过@EventListener(classes =

RemoteApplicationEvent.class)注解修饰。之前已经介绍过该注解，可以将该函数理解为对 RemoteApplicationEvent 事件的监听器，但是在其实现中并非所有的 RemoteApplicationEvent 事件都会处理。根据 if 中的条件，可以看到在该监听处理中，只对事件来源是自己并且事件类型不是 AckRemoteApplicationEvent 的内容进行后续的处理，而后续的处理就是通过消息管道将该事件发送出去。所以，该监听器的功能就是监听本地事件来进行消息的发送。

```
    @EventListener(classes = RemoteApplicationEvent.class)
    public void acceptLocal(RemoteApplicationEvent event) {
        if (this.serviceMatcher.isFromSelf(event)
                && !(event instanceof AckRemoteApplicationEvent)) {
            this.cloudBusOutboundChannel.send(MessageBuilder.withPayload(event).
build());
        }
    }
```

再来看看 acceptRemote 方法。该方法中使用了@StreamListener 注解修饰，该注解的作用是将该函数注册为消息代理上数据流的事件监听器，注解中的属性值 SpringCloudBusClient.INPUT 指定了监听的通道名。同时，回头看该函数所在类的定义，使用了@EnableBinding 注解，该注解用来实现与消息代理的连接，注解中的属性值 SpringCloudBusClient.class 声明了输入和输出通道的定义（这部分内容源自 Spring Cloud Stream，在下一章中，我们会对这些内容做详细介绍，这里我们只需理解它用来绑定消息代理的输入与输出，以实现向消息总线上发送和接收消息即可）。

```
    @StreamListener(SpringCloudBusClient.INPUT)
    public void acceptRemote(RemoteApplicationEvent event) {
        if (event instanceof AckRemoteApplicationEvent) {
            if (this.bus.getTrace().isEnabled() && !this.serviceMatcher.isFromSelf(event)
                    && this.applicationEventPublisher != null) {
                this.applicationEventPublisher.publishEvent(event);
            }
            // If it's an ACK we are finished processing at this point
            return;
        }
        if (this.serviceMatcher.isForSelf(event)
                && this.applicationEventPublisher != null) {
            if (!this.serviceMatcher.isFromSelf(event)) {
                this.applicationEventPublisher.publishEvent(event);
            }
            if (this.bus.getAck().isEnabled()) {
                AckRemoteApplicationEvent ack = new AckRemoteApplicationEvent(this,
                        this.serviceMatcher.getServiceId(),
                        this.bus.getAck().getDestinationService(),
```

```
                event.getDestinationService(), event.getId(), event.getClass());
            this.cloudBusOutboundChannel
                    .send(MessageBuilder.withPayload(ack).build());
            this.applicationEventPublisher.publishEvent(ack);
        }
    }
    if (this.bus.getTrace().isEnabled() && this.applicationEventPublisher != null) {
        // We are set to register sent events so publish it for local consumption,
        // irrespective of the origin
        this.applicationEventPublisher.publishEvent(new SentApplicationEvent(this,
                event.getOriginService(), event.getDestinationService(),
                event.getId(), event.getClass()));
    }
}
```

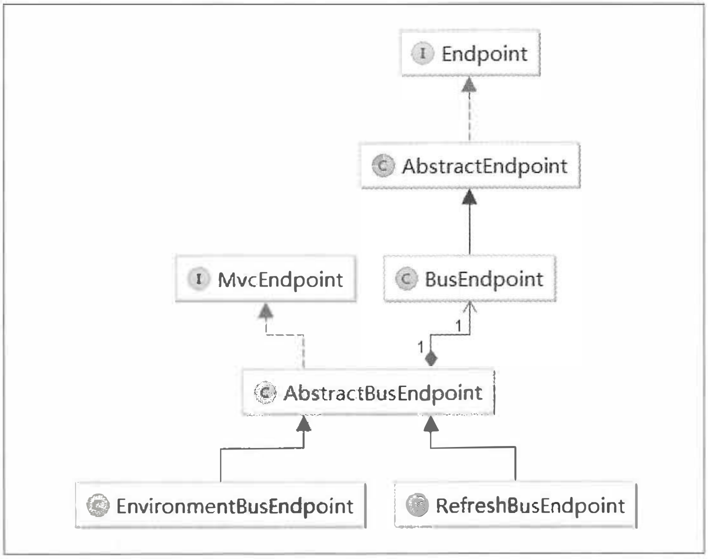
通过上面的分析，我们已经可以知道 Spring Cloud Bus 通过 `acceptRemote` 方法来监听消息代理的输入通道，并根据事件类型和配置内容来确定是否要发布事件给我们之前分析的几个事件监听器来对事件做具体的处理；而 `acceptLocal` 方法用来监听本地的事件，针对事件来源是自己，并且事件类型不是 `AckRemoteApplicationEvent` 的内容通过消息代理的输出通道发送到总线上去。

控制端点

在介绍了 Spring Cloud Bus 中实现的事件模型之后，我们已经知道每个节点是如何响应消息总线上的事件了。那么这些发送到消息总线上用来触发各个节点的事件处理的动作是如何实现的呢？回想一下之前在实现配置属性刷新时，我们在修改了 Git 仓库上的配置信息之后，往总线上的某个节点发送了一个请求`/bus/refresh` 来触发总线上的所有节点进行配置刷新；我们在连接到消息总线的应用启动时，也能在控制台中看到类似下面的输出：

```
    2016-09-30 11:05:13.037  INFO 18720 --- [           main]
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/bus/refresh],methods=[POST]}"
onto public void
org.springframework.cloud.bus.endpoint.RefreshBusEndpoint.refresh(java.lang.String)
    2016-09-30 11:05:13.045  INFO 18720 --- [           main]
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/bus/env],methods=[POST]}" onto
public void
org.springframework.cloud.bus.endpoint.EnvironmentBusEndpoint.env(java.util.Map<jav
a.lang.String, java.lang.String>,java.lang.String)
```

从上面的日志信息中可以看到，在 `org.springframework.cloud.bus. endpoint` 包下的 `RefreshBusEndpoint` 和 `EnvironmentBusEndpoint` 分别创建了两个控制端点：`/bus/refresh` 和`/bus/env`。通过整理 `org.springframework.cloud.bus. endpoint` 包下的内容，我们可以得到如下类图：



从图中可以 发现，Spring Cloud Bus 中的 Endpoint 也是通过 spring-boot-actuator 模块来实现的。下面，简单介绍一下 spring-boot-actuator 模块中的几个重要元素。

- Endpoint：该接口中定义了监控端点需要暴露的一些有用信息，比如，id、是否开启标识、是否开启敏感信息标识等。
- AbstractEndPoint：该抽象类是对 Endpoint 的基础实现，在该抽象类中引入了 Environment 接口 对象，从而对接口暴露信息的控制可以 通过配置文件的方式来控制。
- MvcEndpoint 接口 ：该接口定义了 Endpoint 接口在 MVC 层的策略。在这里可以 通过使用 Spring MVC 的@RequestMapping 注解来定义端点暴露的接口 地址。

下面我们来看看 SpringCloud Bus 是如何扩展 Endpoint 的。

- BusEndpoint：该类继承自 AbstractEndPoint。从类上的注解@Configuration-Properties 配置可以 知道 Spring Cloud Bus 实现的端点配置属性需要以 endpoints.bus 开头， 通过该类的构造函数（配合 AbstractEndpoint 中的构造函数），我们可以 知道默认 id 为 bus，并且端点默认敏感标识为 true：

```
@ConfigurationProperties(prefix = "endpoints.bus", ignoreUnknownFields = false)
public class BusEndpoint extends AbstractEndpoint<Collection<String>> {

    public BusEndpoint() {
        super("bus");
```

```
    }

    @Override
    public Collection<String> invoke() {
        return Collections.emptyList();
    }
}
public abstract class AbstractEndpoint<T> implements Endpoint<T>, EnvironmentAware {
    ...
    public AbstractEndpoint(String id) {
        this(id, true);
    }
    public AbstractEndpoint(String id, boolean sensitive) {
        this.id = id;
        this.sensitiveDefault = sensitive;
    }
    ...
}
```

- AbstractBusEndpoint 类是实现 Spring Cloud Bus 中端点的重要基类，它实现了 MvcEndpoint 接口来暴露 MVC 层的接口，同时关联了 BusEndpoint 对象。通过下面的源码，我们可以看到，getPath、isSensitive 和 getEndpointType 都是委托给 BusEndpoint 来获取的，从而实现通过 Environment 配置接口。

```
public class AbstractBusEndpoint implements MvcEndpoint {

    private ApplicationEventPublisher context;
    private BusEndpoint delegate;
    private String appId;

    public AbstractBusEndpoint(ApplicationEventPublisher context, String appId,
            BusEndpoint busEndpoint) {
        this.context = context;
        this.appId = appId;
        this.delegate = busEndpoint;
    }

    protected String getInstanceId() {
        return this.appId;
    }
```

```
    protected void publish(ApplicationEvent event) {
        context.publishEvent(event);
    }

    @Override
    public String getPath() {
        return "/" + this.delegate.getId();
    }

    @Override
    public boolean isSensitive() {
        return this.delegate.isSensitive();
    }

    @Override
    @SuppressWarnings("rawtypes")
    public Class<? extends Endpoint> getEndpointType() {
        return this.delegate.getClass();
    }
}
```

默认实现的几个端点都继承自 AbstractBusEndpoint 类来实现 MVC 层接口的暴露和配置，下面我们来看看具体的两个实现端点。

- 实现配置刷新的端点 RefreshBusEndpoint 类。通过下面的源码，我们可以看到，在该类中定义了 refresh 的 POST 请求，由于在 BusEndpoint 默认构造时 id 为 bus，而 AbstractBusEndpoint 中 getPath 函数通过 BusEndpoint 中的 id 拼接而成，所以对于 RefreshBusEndpoint 中 refresh 请求的完整路径为 /bus/refresh。同时，该请求通过@RequestParam 注解还定义了一个可选的参数 destination，正如在之前的示例中介绍的，该参数用于指定刷新的服务实例。在请求处理部分直接调用了父类中的 publish 函数将 RefreshRemoteApplicationEvent 事件发布出来，实现在总线上发布消息的功能。

```
public class RefreshBusEndpoint extends AbstractBusEndpoint {

    public RefreshBusEndpoint(ApplicationEventPublisher context, String id,
            BusEndpoint delegate) {
        super(context, id, delegate);
    }
```

```
        @RequestMapping(value = "refresh", method = RequestMethod.POST)
        @ResponseBody
        public void refresh(
                @RequestParam(value = "destination", required = false) String destination) {
            publish(new RefreshRemoteApplicationEvent(this, getInstanceId(),
destination));
        }
    }
```

- EnvironmentBusEndpoint 的实现与 RefreshBusEndpoint 类似，通过暴露 /bus/env 的 POST 请求接口，并提供了 Map 类型的 params 参数设定需要更新的配置信息，以及同 refresh 接口一样的 destination 参数指定需要更新的服务实例，来触发环境参数更新的消息总线控制。

```
    public class EnvironmentBusEndpoint extends AbstractBusEndpoint {

        public EnvironmentBusEndpoint(ApplicationEventPublisher context, String id,
                BusEndpoint delegate) {
            super(context, id, delegate);
        }

        @RequestMapping(value = "env", method = RequestMethod.POST)
        @ResponseBody
        public void env(@RequestParam Map<String, String> params,
                @RequestParam(value = "destination", required = false) String destination) {
            publish(new EnvironmentChangeRemoteApplicationEvent(this, getInstanceId(),
                    destination, params));
        }
    }
```

其他消息代理的支持

由于目前版本的 Spring Cloud Bus 只实现了 RabbitMQ 和 Kafka 的封装，虽然大部分情况下，这两个产品的特性已经涵盖我们大部分的业务场景，但是由于一些特殊需求或是遗留系统等其他因素，有些团队不得不使用其他的消息代理，这个时候我们就需要扩展消息代理的支持。实际上，通过之前对源码的分析，我们可以看到，Spring Cloud Bus 在绑定具体消息代理的输入与输出通道时均使用了抽象接口的方式，所以真正的实现来自于 spring-cloud-starter-bus-amqp 和 spring-cloud-starter-bus-kafka 的依赖。

我们可以查看 spring-cloud-starter-bus-amqp 和 spring-cloud-starter-

bus-kafka 的依赖,可以看到它们分别依赖了 spring-doud-starter-stream-rabbit 和 spring-cloud-starter-stream-kafka真正实现与这些消息代理进行交互操作的是 Spring Cloud Stream。所以，我们在本章中使用的所有 Spring Cloud Bus 的消息通信基础实际上都是由 Spring Cloud Stream 所提供的。一定程度上，我们可以将 Spring Cloud Bus 理解为是一个使用了 Spring Cloud Stream 构建的上层应用。由于 Spring Cloud Stream 为了让开发者屏蔽各个消息代理之间的差异，将来能够方便地切换不同的消息代理而不影响业务程序，所以在业务程序与消息代理之间定义了一层抽象，称为绑定器（Binder）。我们在整合 RabbitMQ 和 Kafka 的时候就是分别引入了它们各自的绑定器实现，可以回想一下之前的实现内容，不论使用 RabbitMQ 还是 Kafka 实现，在程序上其实没有任何变化，变化的只是对绑定器的配置。所以，当我们要在其他消息代理上使用 Spring Cloud Bus 消息总线时，只需要去实现一套指定消息代理的绑定器即可。

第 10 章

消息驱动的微服务：Spring Cloud Stream

Spring Cloud Stream 是一个用来为微服务应用构建消息驱动能力的框架。它可以基于 Spring Boot 来创建独立的、可用于生产的 Spring 应用程序。它通过使用 Spring Integration 来连接消息代理中间件以实现消息事件驱动。Spring Cloud Stream 为一些供应商的消息中间件产品提供了个性化的自动化配置实现，并且引入了发布-订阅、消费组以及分区这三个核心概念。简单地说，Spring Cloud Stream 本质上就是整合了 Spring Boot 和 Spring Integration，实现了一套轻量级的消息驱动的微服务框架。通过使用 Spring Cloud Stream，可以有效简化开发人员对消息中间件的使用复杂度，让系统开发人员可以有更多的精力关注于核心业务逻辑的处理。由于 Spring Cloud Stream 基于 Spring Boot 实现，所以它秉承了 Spring Boot 的优点，自动化配置的功能可帮助我们快速上手使用，但是到目前为止，Spring Cloud Stream 只支持下面两个著名的消息中间件的自动化配置：

- RabbitMQ
- Kafka

对于这两个消息中间件的介绍，我们在上一章消息总线的内容中已有过一些基础的介绍，更多关于这两个消息中间件的高级使用和性能配置等内容不在本书的关注范围之内，读者在实际应用中可以查看它们的官方文档或是其他专业书籍进行进一步学习和实践。

快速入门

下面我们通过构建一个简单的示例来对 Spring Cloud Stream 有一个初步认识。该示例的主要目标是构建一个基于 Spring Boot 的微服务应用，这个微服务应用将通过使用消息中

间件 RabbitMQ 来接收消息并将消息打印到日志中。所以，在进行下面的步骤之前请先确认已经在本地安装了 RabbitMQ，具体安装步骤可根据在上一章中的介绍来操作。

- 创建一个基础的 Spring Boot 工程，命名为 `stream-hello`。
- 编辑 `pom.xml` 中的依赖关系，引入 Spring Cloud Stream 对 RabbitMQ 的支持，具体如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- 创建用于接收来自 RabbitMQ 消息的消费者 SinkReceiver，具体如下：

```
@EnableBinding(Sink.class)
public class SinkReceiver {

    private static Logger logger = LoggerFactory.getLogger(HelloApplication.class);

    @StreamListener(Sink.INPUT)
    public void receive(Object payload) {
        logger.info("Received: " + payload);
    }

}
```

- 创建应用主类，这里同其他 Spring Boot 一样，没有什么特别之处，具体如下：

```
@SpringBootApplication
public class HelloApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloApplication.class, args);
    }

}
```

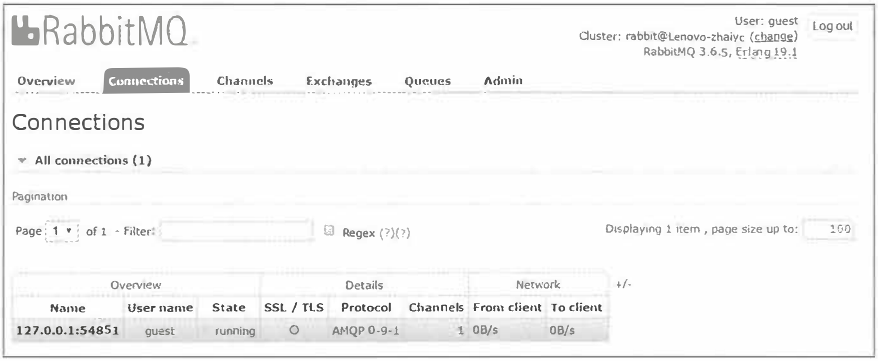
到这里，快速入门示例的编码任务已经完成了。下面我们分别启动 RabbitMQ 以及该 Spring Boot 应用，然后做下面的测试，看看它们是如何运作的。

- 先来看一下 Spring Boot 应用的启动日志。

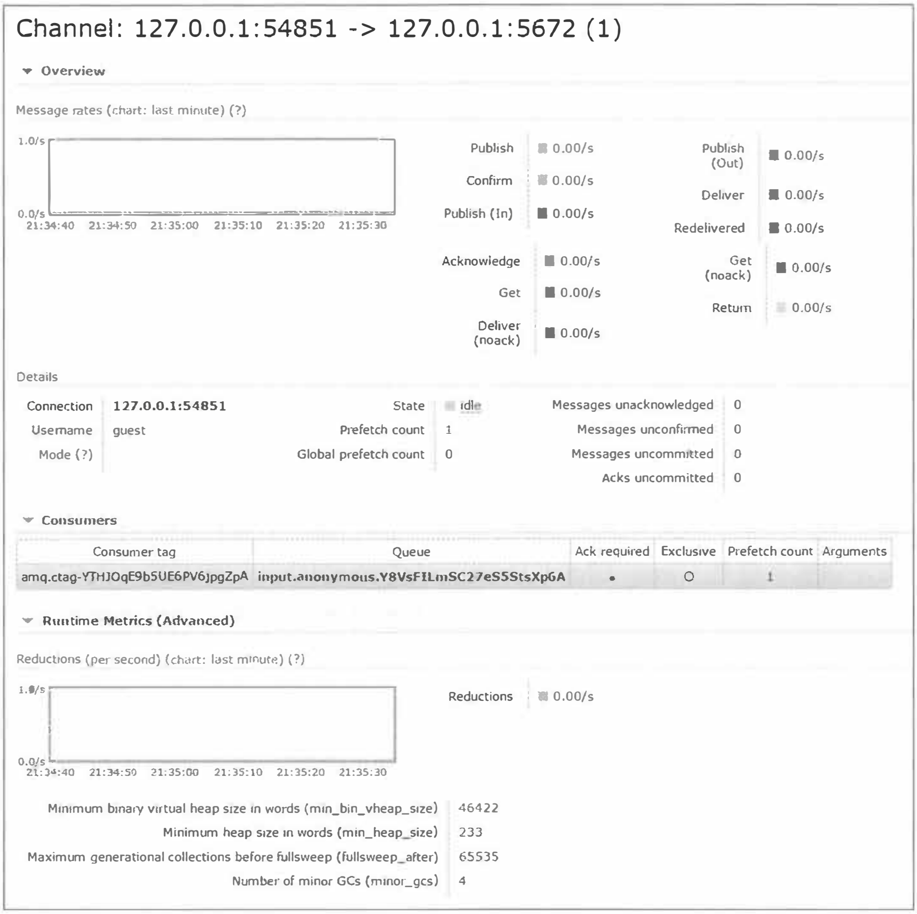
```
    ...
    INFO 16272 --- [main] o.s.c.s.b.r.RabbitMessageChannelBinder   : declaring queue for
inbound: input.anonymous.Y8VsFILmSC27eS5StsXp6A, bound to: input
    INFO 16272 --- [main] o.s.a.r.c.CachingConnectionFactory      : Created new
connection: SimpleConnection@3c78e551 [delegate=amqp://guest@127.0.0.1:5672/]
    INFO 16272 --- [main] o.s.integration.channel.DirectChannel    : Channel
'input.anonymous.Y8VsFILmSC27eS5StsXp6A.bridge' has 1 subscriber(s).
    INFO 16272 --- [main] o.s.i.a.i.AmqpInboundChannelAdapter      : started
inbound.input.anonymous.Y8VsFILmSC27eS5StsXp6A
    ...
```

从上面的日志内容中，可以获得以下信息：

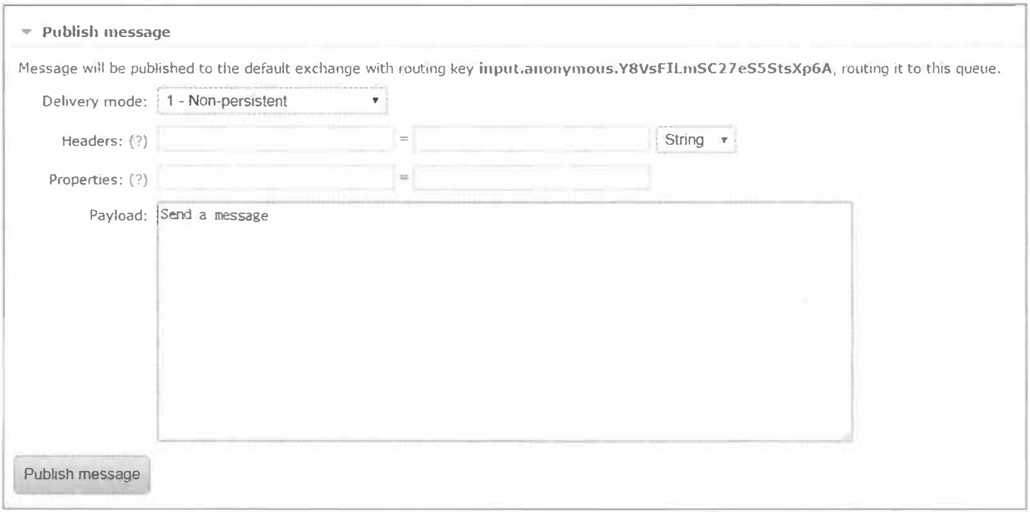
- 使用 guest 用户创建了一个指向 127.0.0.1:5672 位置的 RabbitMQ 连接，在 RabbitMQ 的控制台中我们也可以发现它。



- 声明了一个名为 input.anonymous.Y8VsFILmSC27eS5StsXp6A 的队列，并通过 RabbitMessageChannelBinder 将自己绑定为它的消费者。这些信息我们也能在 RabbitMQ 的控制台中发现它们。



下面我们可以在 RabbitMQ 的控制台中进入 `input.anonymous.Y8VsFILmSC-27eS5StsXp6A` 队列的管理页面，通过 `Publish message` 功能来发送一条消息到该队列中。



此时，我们可以在当前启动的 Spring Boot 应用程序的控制台中看到下面的内容：

```
INFO 16272 --- [C27eS5StsXp6A-1] com.didispace.HelloApplication      : Received:
[B@7cba610e
```

可以发现在应用控制台中输出的内容就是 `SinkReceiver` 中的 `receive` 方法定义的，而输出的具体内容则来自消息队列中获取的对象。这里由于我们没有对消息进行序列化，所以输出的只是该对象的引用，在后面的章节中我们会详细介绍接收消息后的处理。

在顺利完成上面快速入门的示例后，我们简单解释一下上面的步骤是如何将我们的 Spring Boot 应用连接上 RabbitMQ 来消费消息以实现消息驱动业务逻辑的。

首先，我们对 Spring Boot 应用做的就是引入 `spring-cloud-starter-stream-rabbit` 依赖，该依赖包是 Spring Cloud Stream 对 RabbitMQ 支持的封装，其中包含了对 RabbitMQ 的自动化配置等内容。从下面它定义的依赖关系中，我们还可以知道它等价于 `spring-cloud-stream-binder-rabbit` 依赖。

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
    </dependency>
</dependencies>
```

接着，我们再来看看这里用到的几个 Spring Cloud Stream 的核心注解，它们都被定义

在 SinkReceiver 中。

- @EnableBinding，该注解用来指定一个或多个定义了@Input 或@Output 注解的接口，以此实现对消息通道（Channel）的绑定。在上面的例子中，我们通过@EnableBinding(Sink.class)绑定了 Sink 接口，该接口是 Spring Cloud Stream 中默认实现的对输入消息通道绑定的定义，它的源码如下：

```
public interface Sink {

  String INPUT = "input";

  @Input(Sink.INPUT)
  SubscribableChannel input();

}
```

  它通过@Input 注解绑定了一个名为 input 的通道。除了 Sink 之外，Spring Cloud Stream 还默认实现了绑定 output 通道的 Source 接口，还有结合了 Sink 和 Source 的 Processor 接口，实际使用时我们也可以自己通过@Input 和@Output 注解来定义绑定消息通道的接口。当需要为@EnableBinding 指定多个接口来绑定消息通道的时候，可以这样定义：@EnableBinding(value = {Sink.class, Source.class})。

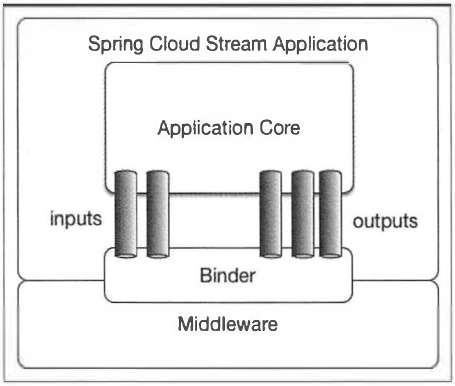
- @StreamListener：如果已经读过上一章 Spring Cloud Bus 的源码分析，相信对该注解不会感到陌生。它主要定义在方法上，作用是将被修饰的方法注册为消息中间件上数据流的事件监听器，注解中的属性值对应了监听的消息通道名。在上面的例子中，我们通过@StreamListener(Sink.INPUT)注解将 receive 方法注册为 input 消息通道的监听处理器，所以当我们在 RabbitMQ 的控制页面中发布消息的时候，receive 方法会做出对应的响应动作。

核心概念

通过上一节介绍的快速入门示例，相信大家对 Spring Cloud Stream 的工作模式已经有了一些基础概念，比如输入、输出通道的绑定，通道消息事件的监听等。在本节中，我们将详细介绍在 Spring Cloud Stream 中是如何通过定义一些基础概念来对各种不同的消息中间件做抽象的。

下图是官方文档中 Spring Cloud Stream 应用模型的结构图。从中我们可以看到，Spring Cloud Stream 构建的应用程序与消息中间件之间是通过绑定器 Binder 相关联的，绑定器对于应用程序而言起到了隔离作用，它使得不同消息中间件的实现细节对应用程序来说是

透明的。所以对于每一个 Spring Cloud Stream 的应用程序来说，它不需要知晓消息中间件的通信细节，它只需知道 Binder 对应程序提供的抽象概念来使用消息中间件来实现业务逻辑即可，而这个抽象概念就是在快速入门中我们提到的消息通道：Channel。如下图所示，在应用程序和 Binder 之间定义了两条输入通道和三条输出通道来传递消息，而绑定器则是作为这些通道和消息中间件之间的桥梁进行通信。



绑定器

Binder 绑定器是 Spring Cloud Stream 中一个非常重要的概念。在没有绑定器这个概念的情况下，Spring Boot 应用要直接与消息中间件进行信息交互的时候，由于各消息中间件构建的初衷不同，所以它们在实现细节上会有较大的差异，这使得我们实现的消息交互逻辑就会非常笨重，因为对具体的中间件实现细节有太重的依赖，当中间件有较大的变动升级或是更换中间件的时候，我们就需要付出非常大的代价来实施。

通过定义绑定器作为中间层，完美地实现了应用程序与消息中间件细节之间的隔离。通过向应用程序暴露统一的 Channel 通道，使得应用程序不需要再考虑各种不同的消息中间件的实现。当需要升级消息中间件，或是更换其他消息中间件产品时，我们要做的就是更换它们对应的 Binder 绑定器而不需要修改任何 Spring Boot 的应用逻辑。这一点在上一章实现消息总线时，从 RabbitMQ 切换到 Kafka 的过程中，已经能够让我们体验到这一好处。

目前版本的 Spring Cloud Stream 为主流的消息中间件产品 RabbitMQ 和 Kafka 提供了默认的 Binder 实现，在快速入门的例子中，我们就使用了 RabbitMQ 的 Binder。另外，Spring Cloud Stream 还实现了一个专门用于测试的 TestSupportBinder，开发者可以直接使用它来对通道的接收内容进行可靠的测试断言。如果要使用除了 RabbitMQ 和 Kafka 以外的消息中间件的话，我们也可以通过使用它所提供的扩展 API 来实现其他中间件的 Binder。

仔细的读者可能已经发现，我们在快速入门示例中，并没有使用 application.

properties 或是 application.yml 来做任何属性设置。那是因为它也秉承了 Spring Boot 的设计理念，提供了对 RabbitMQ 默认的自动化配置。当然，我们也可以通过 Spring Boot 应用支持的任何方式来修改这些配置，比如，通过应用程序参数、环境变量、application.properties 或是 application.yml 配置文件等。比如，下面就是通过配置文件来对 RabbitMQ 的连接信息以及 input 通道的主题进行配置的示例：

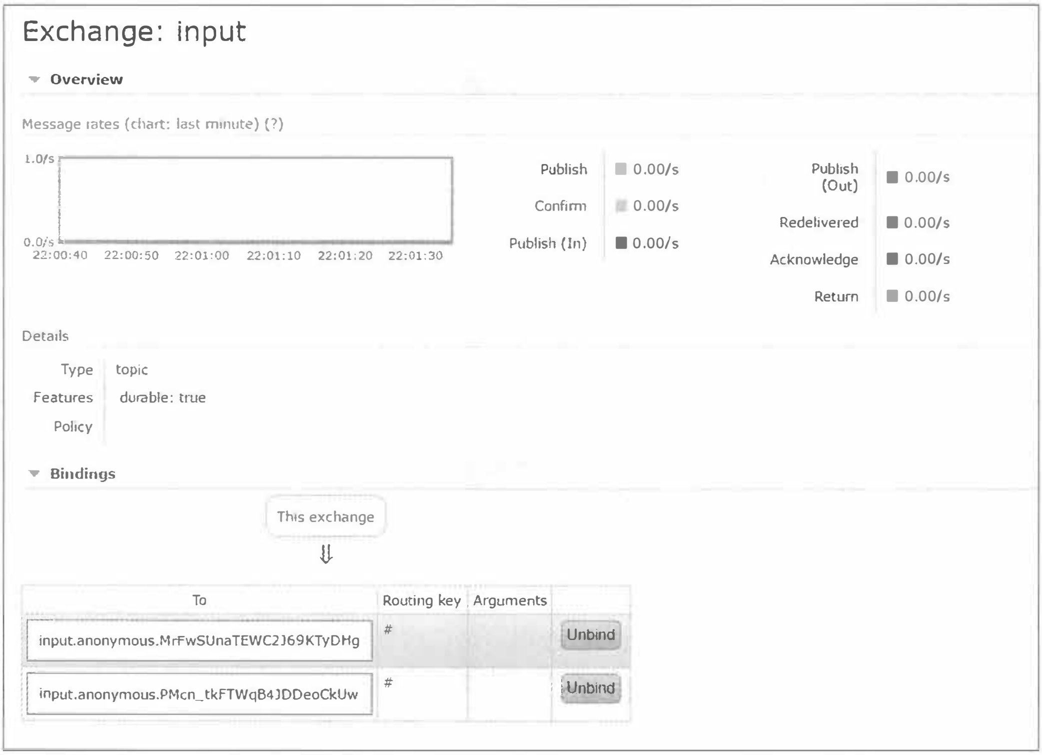
```
spring.cloud.stream.bindings.input.destination=raw-sensor-data

spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=springcloud
spring.rabbitmq.password=123456
```

发布-订阅模式

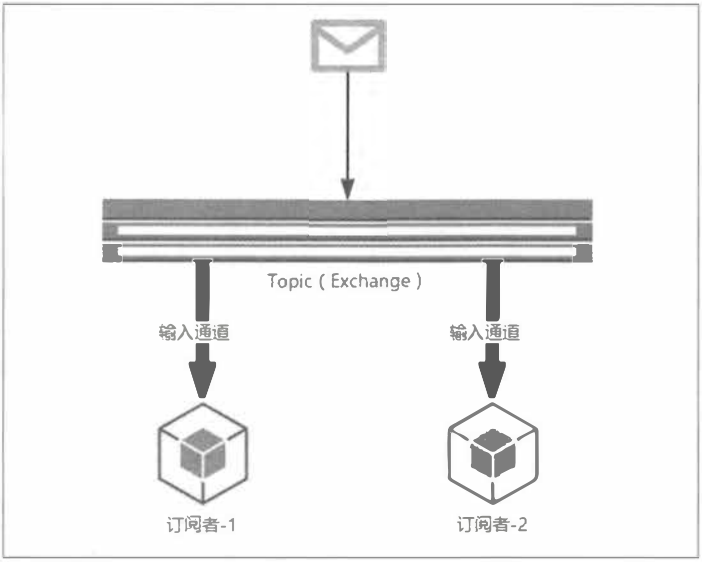
Spring Cloud Stream 中的消息通信方式遵循了发布-订阅模式，当一条消息被投递到消息中间件之后，它会通过共享的 Topic 主题进行广播，消息消费者在订阅的主题中收到它并触发自身的业务逻辑处理。这里所提到的 Topic 主题是 Spring Cloud Stream 中的一个抽象概念，用来代表发布共享消息给消费者的地方。在不同的消息中间件中，Topic 可能对应不同的概念，比如，在 RabbitMQ 中，它对应 Exchange，而在 Kakfa 中则对应 Kafka 中的 Topic。

在快速入门的示例中，我们通过 RabbitMQ 的 Ch annel 发布消息给我们编写的应用程序消费，而实际上 Spring Cloud Stream 应用启动的时候，在 RabbitMQ 的 Exchange 中也创建了一个名为 input 的 Exchange 交换器，由于 Binder 的隔离作用，应用程序并无法感知它的存在，应用程序只知道自己指向 Binder 的输入或是输出通道。为了直观地感受在发布-订阅模式中，消息是如何被分发到多个订阅者的，我们可以使用快速入门的例子，通过命令行的方式启动两个不同端口的进程。此时，我们在 RabbitMQ 控制页面的 Channels 选项卡中看到如下图所示的两个消息通道，它们分别绑定了启动的两个应用程序。

而在 Exchanges 选项卡中，我们还能找到名为 `input` 的交换器，单击进入可以看到如下图所示的详情页面。Bindings 栏中的内容就是两个应用程序绑定通道中的消息队列，我们可以通过 Exchange 页面的 Publish Message 来发布消息，此时可以发现两个启动的应用程序都输出了消息内容。



下图总结了我们上面所做尝试的基础结构。启动的两个应用程序分别是“订阅者-1”和“订阅者-2”，它们都建立了一条输入通道绑定到同一个 `Topic`（RabbitMQ 的 Exchange）上。当该 `Topic` 中有消息发布进来后，连接到该 `Topic` 上的所有订阅者可以收到该消息并根据自身的需求进行消费操作。

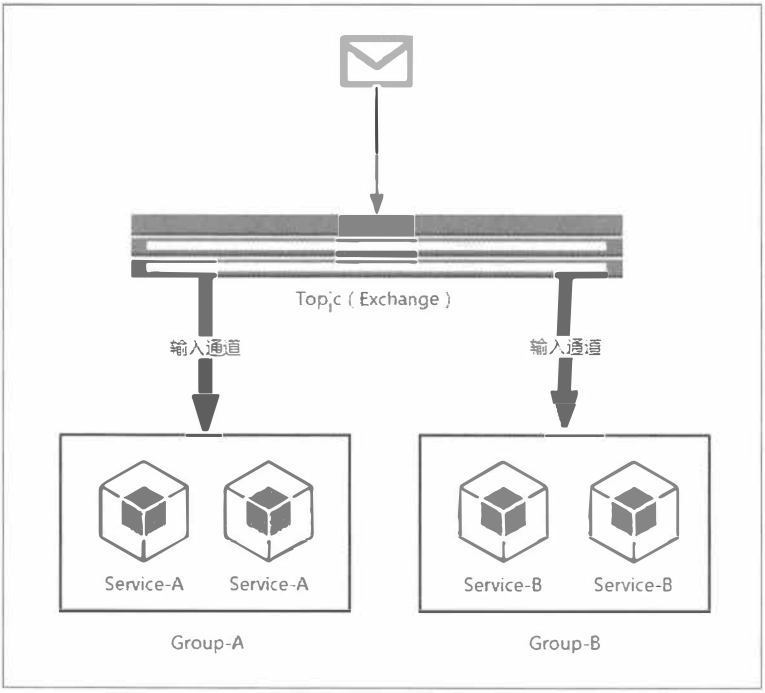


相对于点对点队列实现的消息通信来说，Spring Cloud Stream 采用的发布-订阅模式可以有效降低消息生产者与消费者之间的耦合。当需要对同一类消息增加一种处理方式时，只需要增加一个应用程序并将输入通道绑定到既有的 Topic 中就可以实现功能的扩展，而不需要改变原来已经实现的任何内容。

消费组

虽然 Spring Cloud Stream 通过发布-订阅模式将消息生产者与消费者做了很好的解耦，基于相同主题的消费者可以轻松地进行扩展，但是这些扩展都是针对不同的应用实例而言的。在现实的微服务架构中，我们的每一个微服务应用为了实现高可用和负载均衡，实际上都会部署多个实例。在很多情况下，消息生产者发送消息给某个具体微服务时，只希望被消费一次，按照上面我们启动两个应用的例子，虽然它们同属一个应用，但是这个消息出现了被重复消费两次的情况。为了解决这个问题，在 Spring Cloud Stream 中提供了消费组的概念。

如果在同一个主题上的应用需要启动多个实例的时候，我们可以通过 spring.cloud.stream.bindings.input.group 属性为应用指定一个组名，这样这个应用的多个实例在接收到消息的时候，只会有一个成员真正收到消息并进行处理。如下图所示，我们为 Service-A 和 Service-B 分别启动了两个实例，并且根据服务名进行了分组，这样当消息进入主题之后，Group-A 和 Group-B 都会收到消息的副本，但是在两个组中都只会有一个实例对其进行消费。



默认情况下，当没有为应用指定消费组的时候，Spring Cloud Stream 会为其分配一个独立的匿名消费组。所以，如果同一主题下的所有应用都没有被指定消费组的时候，当有消息发布之后，所有的应用都会对其进行消费，因为它们各自都属于一个独立的组。大部分情况下，我们在创建 Spring Cloud Stream 应用的时候，建议最好为其指定一个消费组，以防止对消息的重复处理，除非该行为需要这样做（比如刷新所有实例的配置等）。

消息分区

通过引入消费组的概念，我们已经能够在多实例的情况下，保障每个消息只被组内的一个实例消费。通过上面对消费组参数设置后的实验，我们可以观察到，消费组无法控制消息具体被哪个实例消费。也就是说，对于同一条消息，它多次到达之后可能是由不同的实例进行消费的。但是对于一些业务场景，需要对一些具有相同特征的消息设置每次都被同一个消费实例处理，比如，一些用于监控服务，为了统计某段时间内消息生产者发送的报告内容，监控服务需要在自身聚合这些数据，那么消息生产者可以为消息增加一个固有的特征 ID 来进行分区，使得拥有这些 ID 的消息每次都能被发送到一个特定的实例上实现累计统计的效果，否则这些数据就会分散到各个不同的节点导致监控结果不一致的情况。而分区概念的引入就是为了解决这样的问题：当生产者将消息数据发送给多个消费者实例时，保证拥有共同特征的消息数据始终是由同一个消费者实例接收和处理。

Spring Cloud Stream 为分区提供了通用的抽象实现，用来在消息中间件的上层实现分区处理，所以它对于消息中间件自身是否实现了消息分区并不关心，这使得 Spring Cloud Stream 为不具备分区功能的消息中间件也增加了分区功能扩展。

使用详解

在介绍了 Spring Cloud Steam 的基础结构和核心概念之后，下面我们来详细地学习一下它所提供的一些核心注解的具体使用方法。

开启绑定功能

在 Spring Cloud Stream 中，我们需要通过`@EnableBinding`注解来为应用启动消息驱动的功能，该注解我们在快速入门中已经有了基本的介绍，下面来详细看看它的定义：

```
@Target({ElementType.TYPE, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@Import({
 ChannelBindingServiceConfiguration.class,
 BindingBeansRegistrar.class,
 BinderFactoryConfiguration.class,
 SpelExpressionConverterConfiguration.class})
@EnableIntegration
public @interface EnableBinding {

   Class<?>[] value() default {};

}
```

从该注解的定义中我们可以看到，它自身包含了`@Configuration`注解，所以用它注解的类也会成为 Spring 的基本配置类。另外该注解还通过`@Import`加载了 Spring Cloud Stream 运行需要的几个基础配置类。

- `ChannelBindingServiceConfiguration`：该配置会加载消息通道绑定必要的一些实例，比如，用于处理消息通道绑定的 `ChannelBindingService` 实例、消息类型转换器 `MessageConverterConfigurer`、消息通道工厂 `BindableChannel-Factory` 等重要实例，有兴趣的读者可以自行查看这些默认配置，以对其有更深入的理解。

- BindingBeansRegistrar：该类是 ImportBeanDefinitionRegistrar 接口的实现，主要是在 Spring 加载 Bean 的时候被调用，用来实现加载更多的 Bean。由于 BindingBeansRegistrar 被@EnableBinding 注解的@Import 所引用，所以在其他配置加载完后，它的实现会被回调来创建其他的 Bean，而这些 Bean 则从@EnableBinding 注解的 value 属性定义的类中获取。就如我们入门实例中定义的@EnableBinding(Sink.class)，它在加载用于消息驱动的基础 Bean 之后，会继续加载 Sink 中定义的具体消息通道绑定。
- BinderFactoryConfiguration：Binder 工厂的配置，主要用来加载与消息中间件相关的配置信息，比如，它会从应用工程的 META-INF/spring.binders 中加载针对具体消息中间件相关的配置文件等。
- SpelExpressionConverterConfiguration：SpEL 表达式转换器配置。

@EnableBinding 注解只有一个唯一的属性：value。上面已经介绍过，由于该注解@Import 了 BindingBeansRegistrar 实现，所以在加载了基础配置内容之后，它会回调来读取 value 中的类，以创建消息通道的绑定。另外，由于 value 是一个 Class 类型的数组，所以我们可以通过 value 属性一次性指定多个关于消息通道的配置。

绑定消息通道

在 Spring Cloud Steam 中，我们可以在接口中通过@Input 和@Output 注解来定义消息通道，而用于定义绑定消息通道的接口则可以被@EnableBinding 注解的 value 参数来指定，从而在应用启动的时候实现对定义消息通道的绑定。

在快速入门的示例中，我们演示了使用 Sink 接口绑定的消息通道。Sink 接口是 Spring Cloud Steam 提供的一个默认实现，除此之外还有 Source 和 Processor，可从它们的源码中学习它们的定义方式：

```
public interface Sink {

    String INPUT = "input";

    @Input(Sink.INPUT)
    SubscribableChannel input();

}

public interface Source {

    String OUTPUT = "output";

    @Output(Source.OUTPUT)
```

```
    MessageChannel output();

}

public interface Processor extends Source, Sink {

}
```

从上面的源码中，我们可以看到，Sink 和 Source 中分别通过@Input 和@Output 注解定义了输入通道和输出通道，而 Processor 通过继承 Source 和 Sink 的方式同时定义了一个输入通道和一个输出通道。

另外，@Input 和@Output 注解都还有一个 value 属性，该属性可以用来设置消息通道的名称，这里 Sink 和 Source 中指定的消息通道名称分别为 input 和 output。如果我们直接使用这两个注解而没有指定具体的 value 值，将默认使用方法名作为消息通道的名称。

最后，需要注意一点，当我们定义输出通道的时候，需要返回 MessageChannel 接口对象，该接口定义了向消息通道发送消息的方法；而定义输入通道时，需要返回 SubscribableChannel 接口对象，该接口继承自 MessageChannel 接口，它定义了维护消息通道订阅者的方法。

注入绑定接口

在完成了消息通道绑定的定义之后，Spring Cloud Stream 会为其创建具体的实例，而开发者只需要通过注入的方式来获取这些实例并直接使用即可。举个简单的例子，我们在快速入门示例中已经为 Sink 接口绑定的 input 消息通道实现了具体的消息消费者，下面可以通过注入的方式实现一个消息生成者，向 input 消息通道发送数据。

- 创建一个将 Input 消息通道作为输出通道的接口，具体如下：

```
public interface SinkSender {

    @Output(Sink.INPUT)
    MessageChannel output();

}
```

- 对快速入门中定义的 SinkReceiver 做一些修改：在@EnableBinding 注解中增加对 SinkSender 接口的指定，使 Spring Cloud Stream 能创建出对应的实例。

```
@EnableBinding(value = {Sink.class, SinkSender.class})
public class SinkReceiver {
```

```
    private static Logger logger = LoggerFactory.getLogger(SinkReceiver.class);

    @StreamListener(Sink.INPUT)
    public void receive(Object payload) {
        logger.info("Received: " + payload);
    }

}
```

- 创建一个单元测试类，通过@Autowired 注解注入 SinkSender 的实例，并在测试用例中调用它的发送消息方法。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = HelloApplication.class)
@WebAppConfiguration
public class HelloApplicationTests {

    @Autowired
    private SinkSender sinkSender;

    @Test
    public void contextLoads() {
        sinkSender.output().send(MessageBuilder.withPayload("From
SinkSender").build());
    }

}
```

- 运行该单元测试用例，如果可以在控制台中找到如下输出内容，表明我们的试验已经成功了，消息被正确地发送到了 input 通道中，并被相对应的消息消费者输出。

```
...
INFO 10656 --- [           main] com.didispace.HelloApplication         : Received:
From SinkSender
...
```

注入消息通道

由于 Spring Cloud Stream 会根据绑定接口中的@Input 和@Output 注解来创建消息通道实例，所以我们也可以通过直接注入的方式来使用消息通道对象。比如，我们可以通过下面的示例，注入上面例子中 SinkSender 接口中定义的名为 input 的消息输入通道。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = HelloApplication.class)
@WebAppConfiguration
public class HelloApplicationTests {
```

```
    @Autowired
    private MessageChannel input;

    @Test
    public void contextLoads() {
        input.send(MessageBuilder.withPayload("From MessageChannel").build());
    }

}
```

上面定义的内容，完成了与之前通过注入绑定接口 SinkSender 方式实现的测试用例相同的操作。因为在通过注入绑定接口实现时，sinkSender.output()方法实际获得的就是 SinkSender 接口中定义的 MessageChannel 实例，只是在这里我们直接通过注入的方式来实现了而已。这种用法虽然很直接，但是也容易犯错，很多时候我们在一个微服务应用中可能会创建多个不同名的 MessageChannel 实例，这样通过@Autowired 注入时，要注意参数命名需要与通道同名才能被正确注入，或者也可以使用@Qualifier 注解来特别指定具体实例的名称，该名称需要与定义 MessageChannel 的@Output 中的 value 参数一致，这样才能被正确注入。比如下面的例子，在一个接口中定义了两个输出通道，分别命名为 Output-1 和 Output-2，当要使用 Output-1 的时候，可以通过@Qualifier("Output-1")来指定这个具体的实例来注入使用。

```
public interface MySource {

    @Output("Output-1")
    MessageChannel output1();

    @Output("Output-2")
    MessageChannel output2();

}

@Component
public class OutputSender {

    @Autowired @Qualifier("Output-1")
    private MessageChannel output;
    ...

}
```

消息生产与消费

由于 Spring Cloud Stream 是基于 Spring Integration 构建起来的，所以在使用 Spring Cloud Stream 构建消息驱动服务的时候，完全可以使用 Spring Integration 的原生注解来实现各种业务需求。同时，为了简化面向消息的编程模型，Spring Cloud Stream 还提供了 `@StreamListener` 注解对输入通道的处理做了进一步优化。下面我们分别从这两方面来学习一下对消息的处理。

Spring Integration 原生支持

通过之前的内容，我们已经能够通过注入绑定接口和消息通道的方式实现向名为 `input` 的消息通道发送信息。接下来，我们通过 Spring Integration 原生的`@ServiceActivator` 和`@InboundChannelAdapter` 注解来尝试实现相同的功能，具体实现如下：

```
@EnableBinding(value = {Sink.class})
public class SinkReceiver {

    private static Logger logger = LoggerFactory.getLogger(SinkReceiver.class);

    @ServiceActivator(inputChannel=Sink.INPUT)
    public void receive(Object payload) {
        logger.info("Received: " + payload);
    }

}

@EnableBinding(value = {SinkSender.SinkOutput.class})
public class SinkSender {

    private static Logger logger = LoggerFactory.getLogger(SinkSender.class);

    @Bean
    @InboundChannelAdapter(value = SinkOutput.OUTPUT, poller = @Poller(fixedDelay
= "2000"))
    public MessageSource<Date> timerMessageSource() {
        return () -> new GenericMessage<>(new Date());
    }

    public interface SinkOutput {

        String OUTPUT = "input";

        @Output(SinkOutput.OUTPUT)
```

```
        MessageChannel output();
    }

}
```

上面展示的两段代码分别属于两个不同的应用。

- SinkReceiver 类属于消息消费者实现，与之前实现的类似，只是做了一些修改：使用原生的@ServiceActivator 注解替换了@StreamListener，实现对 Sink.INPUT 通道的监听处理，而该通道绑定了名为 input 的主题。
- SinkSender 类属于消息生产者实现，它在内部定义了 SinkOutput 接口来将输出通道绑定到名为 input 的主题中。由于 SinkSender 和 SinkReceiver 共用一个主题，所以它们构成了一组生产者与消费者。另外，在 SinkSender 中还创建了用于生产消息的 timerMessageSource 方法，该方法会将当前时间作为消息返回。而@InboundChannelAdapter 注解定义了该方法是对 SinkOutput.OUTPUT 通道的输出绑定，同时使用 poller 参数将该方法设置为轮询执行，这里我们定义为2000毫秒，所以它会以2秒的频率向 SinkOutput.OUTPUT 通道输出当前时间。

执行上面定义的程序，可以得到类似下面的输出：

```
    INFO 248 --- [ask-scheduler-2] com.didispace.HelloApplication          : Received:
Sun Nov 13 11:22:39 CST 2016
    INFO 248 --- [ask-scheduler-3] com.didispace.HelloApplication          : Received:
Sun Nov 13 11:22:41 CST 2016
    INFO 248 --- [ask-scheduler-2] com.didispace.HelloApplication          : Received:
Sun Nov 13 11:22:43 CST 2016
    INFO 248 --- [ask-scheduler-1] com.didispace.HelloApplication          : Received:
Sun Nov 13 11:22:45 CST 2016
    INFO 248 --- [ask-scheduler-5] com.didispace.HelloApplication          : Received:
Sun Nov 13 11:22:47 CST 2016
```

另外，还可以通过@Transformer 注解对指定通道的消息进行转换。比如，我们可以在上面的 SinkSender 类中增加下面的内容：

```
    @Transformer(inputChannel = Sink.INPUT, outputChannel = Sink.INPUT)
    public Object transform(Date message) {
        return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(message);
    }
```

再次执行程序，可以得到下面的输出内容，消息内容被成功转换为 yyyy-MM-dd HH:mm:ss 格式了。

```
    INFO 22092 --- [ask-scheduler-2] com.didispace.HelloApplication         : Received:
2016-11-13 12:12:21
    INFO 22092 --- [ask-scheduler-3] com.didispace.HelloApplication         : Received:
2016-11-13 12:12:23
```

```
    INFO 22092 --- [ask-scheduler-2] com.didispace.HelloApplication          : Received:
2016-11-13 12:12:25
    INFO 22092 --- [ask-scheduler-1] com.didispace.HelloApplication          : Received:
2016-11-13 12:12:27
    INFO 22092 --- [ask-scheduler-5] com.didispace.HelloApplication          : Received:
2016-11-13 12:12:29
```

更多关于 Spring Integration 的使用细节请查阅官方文档。

`@StreamListener` 详解

通过入门示例，对于`@StreamListener`注解，大家应该都已经有了一些基本的认识，通过该注解修饰的方法，Spring Cloud Steam 会将其注册为输入消息通道的监听器。当输入消息通道中有消息到达的时候，会立即触发该注解修饰方法的处理逻辑对消息进行消费。

消息转换

`@SteamListener` 和`@ServiceActivator`注解都实现了对输入消息通道的监听，但是`@SteamListener`相比`@ServiceActivator`更为强大，因为它还内置了一系列的消息转换功能，这使得基于`@SteamListener`注解实现的消息处理模型更为简单。

大部分情况下，我们通过消息来对接服务或系统时，消息生产者都会以结构化的字符串形式来发送，比如 JSON 或 XML。当消息到达的时候，输入通道的监听器需要对该字符串做一定的转化，将 JSON 或 XML 转换成具体的对象，然后再做后续的处理。

假设，我们需要传输一个 User 对象，该对象有 `name` 和 `age` 两个字段，这时，如果使用`@ServiceActivator`注解的话，可以通过下面的代码实现：

```
@EnableBinding(value = {Sink.class})
public class SinkReceiver {

    private static Logger logger = LoggerFactory.getLogger(SinkReceiver.class);

    @ServiceActivator(inputChannel=Sink.INPUT)
    public void receive(User user) {
        logger.info("Received: " + user);
    }

    @Transformer(inputChannel = Sink.INPUT, outputChannel = Sink.INPUT)
    public User transform(String message) throws Exception {
        ObjectMapper objectMapper = new ObjectMapper();
        User user = objectMapper.readValue(message, User.class);
        return user;
    }

}
```

```
@EnableBinding(value = {SinkSender.SinkOutput.class})
public class SinkSender {

    private static Logger logger = LoggerFactory.getLogger(SinkSender.class);

    @Bean
    @InboundChannelAdapter(value = Sink.INPUT, poller = @Poller(fixedDelay = "2000"))
    public MessageSource<String> timerMessageSource() {
        return () -> new GenericMessage<>("{\"name\":\"didi\", \"age\":30}");
    }

    public interface SinkOutput {

        String OUTPUT = "input";

        @Output(SinkOutput.OUTPUT)
        MessageChannel output();

    }
}
```

由于`@ServiceActivator`本身不具备对消息的转换能力，所以当代表 User 对象的 JSON 字符串到达后，它自身无法将其转换成 User 对象。所以，这里需要通过`@Transformer`注解帮助将字符串类型的消息转换成 User 对象，并将转换结果传递给`@ServiceActivator`的处理方法做后续的消费。

如果我们使用`@SteamListener`注解的话，就可以把上面的实现简化为下面的代码：

```
@EnableBinding(value = {Sink.class})
public class SinkReceiver {

    private static Logger logger = LoggerFactory.getLogger(SinkReceiver.class);

    @StreamListener(Sink.INPUT)
    public void receive(User user) {
        logger.info("Received: " + user);
    }

}
```

从上面的实现中可以看到，我们去掉了`@Transformer`注解的方法，同时用`@StreamListener`替代了`@ServiceActivator`注解，而`SinkSender`类不需要做任何修改，只需在配置文件中增加`spring.cloud.stream.bindings.input.content-type=application/json`属性设置，这样我们可以得到与之前一样的结果。

@StreamListener 注解能够通过配置属性实现 JSON 字符串到对象的转换，这是因为在 Spring Cloud Steam 中实现了一套可扩展的消息转换机制。在消息消费逻辑执行之前，消息转换机制会根据消息头信息中声明的消息类型（即上面对 input 通道配置的 content-type 参数），找到对应的消息转换器并实现对消息的自动转换。

消息反馈

很多时候在处理完输入消息之后，需要反馈一个消息给对方，这时候可以通过 @SendTo 注解来指定返回内容的输出通道。

比如，假设我们有这样的两个应用，App1 和 App2。

- App1 中实现了对 input 输入通道的监听，并且在接收到消息之后，对消息做了一些简单加工，然后通过@SendTo 把处理方法返回的内容以消息的方式发送到 output 通道中，具体如下：

```
@EnableBinding(value = {Processor.class})
public class App1 {

    private static Logger logger = LoggerFactory.getLogger(App1.class);

    @StreamListener(Processor.INPUT)
    @SendTo(Processor.OUTPUT)
    public Object receiveFromInput(Object payload) {
        logger.info("Received: " + payload);
        return "From Input Channel Return - " + payload;
    }

}
```

- App2 是 App1 应用中 input 通道的生产者以及 output 通道的消费者。为了继续使用 Processor 绑定接口的定义，我们可以在配置文件中将该应用的 input 和 output 绑定反向地做一些配置。因为对于 App2 来说，它的 input 绑定通道实际上是对 output 主题的消费者，而 output 绑定通道实际上是对 input 主题的生产者，所以我们可以做如下具体配置。指定通道的具体主题来实现与 App1 应用的消息交互：

```
spring.cloud.stream.bindings.input.destination=output
spring.cloud.stream.bindings.output.destination=input
```

- App2 中程序的逻辑实现如下所示。通过 timerMessageSource 方法，向 output 绑定的输出通道中发送内容，也就是向名为 input 的主题中传递消息，这样 App1 中的 input 输入通道就可以收到这个消息。同时，这里还创建了对 input 输入消

息通道的绑定。通过上面的配置，它会监听来自 output 主题的消息，通过 receiveFromOutput 方法，会将消息内容输出。

```
@EnableBinding(value = {Processor.class})
public class App2 {

    private static Logger logger = LoggerFactory.getLogger(App2.class);

    @Bean
    @InboundChannelAdapter(value = Processor.OUTPUT, poller = @Poller(fixedDelay =
"2000"))
    public MessageSource<Date> timerMessageSource() {
        return () -> new GenericMessage<>(new Date());
    }

    @StreamListener(Processor.INPUT)
    public void receiveFromOutput(Object payload) {
        logger.info("Received: " + payload);
    }

}
```

将 App1 和 App2 同时运行起来，我们分别可以从它们的控制台中看到如下内容。

- App1 应用的控制台输出如下所示，它输出了来自 App2 应用中定义的向 input 主题中轮询发送的时间。

```
INFO 28180 --- [mWYODqFPjxMmg-1] com.didispace.HelloApplication      : Received:
Wed Nov 16 08:19:16 CST 2016
INFO 28180 --- [mWYODqFPjxMmg-1] com.didispace.HelloApplication      : Received:
Wed Nov 16 08:19:18 CST 2016
INFO 28180 --- [mWYODqFPjxMmg-1] com.didispace.HelloApplication      : Received:
Wed Nov 16 08:19:20 CST 2016
```

- App2 应用的控制台输出如下所示，由于 App1 应用在打印了 input 主题中的消息之后做了一些简单的字符串拼接，然后将拼接后的字符串输出到 output 主题，而下面的输出内容正是 App2 应用从 output 主题中获取的消息内容。

```
INFO 31916 --- [xmrMN07yhd8Ag-1] com.didispace.HelloApplication      : Received:
From Input Channel Return - Wed Nov 16 08:19:16 CST 2016
INFO 31916 --- [xmrMN07yhd8Ag-1] com.didispace.HelloApplication      : Received:
From Input Channel Return - Wed Nov 16 08:19:18 CST 2016
INFO 31916 --- [xmrMN07yhd8Ag-1] com.didispace.HelloApplication      : Received:
From Input Channel Return - Wed Nov 16 08:19:20 CST 2016
```

在 Spring Cloud Stream 中除了可以使用@SendTo 注解将方法返回结果输出到消息通道中，还可以使用原生注解@ServiceActivator 的 outputChannel 属性配置输出通道

把返回结果发送给消息中间件。根据上面的例子，我们可以将 App1 应用修改成下面这样，它们实现的效果是一样的。

```
@EnableScheduling
@EnableBinding(value = {Processor.class})
public class App1 {

    private static Logger logger = LoggerFactory.getLogger(App1.class);

    @ServiceActivator(inputChannel= Processor.INPUT, outputChannel =
Processor.OUTPUT)
    public Object receiveFromInput(Object payload) {
        logger.info("Received: " + payload);
        return "From Input Channel Return - " + payload;
    }

}
```

响应式编程

在 Spring Cloud Stream 中还支持使用基于 RxJava 的响应式编程来处理消息的输入和输出。与 RxJava 的整合使用同样很容易，下面我们详细看看如何使用 RxJava 实现上面消息反馈中试验的场景：App1 应用接收来自 App2 应用发送到 `input` 主题的消息，并返回一条消息到 `output` 主题供 App2 应用消息输出。

- 在 `pom.xml` 中引入 `spring-cloud-stream-rxjava` 依赖，具体如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-rxjava</artifactId>
    <version>1.0.2.RELEASE</version>
</dependency>
```

- 改造 App1 的实现，具体如下：

```
@EnableRxJavaProcessor
public class App1 {

    private static Logger logger = LoggerFactory.getLogger(App1.class);

    @Bean
    public RxJavaProcessor<String,String> processor() {
        return inputStream -> inputStream.map(data -> {
            logger.info("Received: " + data);
            return data;
```

```
        }).map(data -> String.valueOf("From Input Channel Return - " + data));
    }

}
```

通过上面的改造，我们再次运行 App1 和 App2 就可以得到与上一节消息反馈中的试验相同的结果。下面对前面的实现内容做一些说明。

- 在 App1 的类名上，我们使用 `@EnableRxJavaProcessor` 替代了原来的 `@EnableBinding(value = {Processor.class})`，该注解用来标识当前类中应该提供一个 `RxJavaProcessor` 实现的 Bean。另外从其源码中还可以看到它自身就实现了对 `Processor` 定义通道的绑定：`@EnableBinding({Processor.class})`。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@EnableBinding({Processor.class})
@Import({RxJavaProcessorConfiguration.class})
public @interface EnableRxJavaProcessor {
}
```

- 在 App1 中还定义了一个 `RxJavaProcessor` 的实现 Bean。在 `RxJavaProcessor` 接口中定义了一个用来处理输入通道和返回内容给输出通道的 `process` 方法，由于输入输出都采用了 `Observable`，所以该方法只会在应用启动的时候调用一次，用来设置数据流。当有消息到达输入通道时，会采用 RxJava 实现的观察者模式来消费和输出内容。

```
public interface RxJavaProcessor<I, O> {

    Observable<O> process(Observable<I> input);

}
```

- 除了实现上面的场景之外，通过利用 RxJava 的支持，我们还能轻易地实现消息的缓存聚合。比如，我们希望 App1 在接收到 5 条消息之后才将处理结果返回给输出通道，那么只需通过下面的改进即可轻松实现这样的场景：

```
@EnableRxJavaProcessor
public class App1 {

    private static Logger logger = LoggerFactory.getLogger(App1.class);

    @Bean
    public RxJavaProcessor<String,String> processor() {
```

```
        return inputStream -> inputStream.map(data -> {
            logger.info("Received: " + data);
            return data;
        }).buffer(5).map(data -> String.valueOf("From Input Channel Return - " +
data));
    }

}
```

再次执行 App1 和 App2 应用，此时我们在 App1 应用中会得到一样的输出，但是在 App2 应用中会得到下面的输出：

```
INFO 8796 --- [-maOhv-th7Raw-1] com.didispace.HelloApplication          : Received:
From Input Channel Return - [1479390916408, 1479390918409, 1479390920412, 1479390922414,
1479390924415]
INFO 8796 --- [-maOhv-th7Raw-1] com.didispace.HelloApplication          : Received:
From Input Channel Return - [1479390926416, 1479390928418, 1479390930419, 1479390932421,
1479390934423]
```

消费组与消息分区

在“核心概念”一节中，我们对消费组和消息分区已经进行了基本的介绍，在这里来详细介绍一下这两个概念的使用方法。

消费组

通常每个服务都不会以单节点的方式运行在生产环境中，当同一个服务启动多个实例的时候，这些实例会绑定到同一个消息通道的目标主题上。默认情况下，当生产者发出一条消息到绑定通道上，这条消息会产生多个副本被每个消费者实例接收和处理。但是在有些业务场景之下，我们希望生产者产生的消息只被其中一个实例消费，这个时候就需要为这些消费者设置消费组来实现这样的功能。实现的方式非常简单，只需在服务消费者端设置 `spring.cloud.stream.bindings.input.group` 属性即可，比如可以像下面这样实现。

- 可以先实现一个消费者应用 `SinkReceiver`，实现 `greetings` 主题上的输入通道绑定，它的实现如下：

```
@EnableBinding(value = {Sink.class})
public class SinkReceiver {

    private static Logger logger = LoggerFactory.getLogger(SinkReceiver.class);

    @StreamListener(Sink.INPUT)
    public void receive(User user) {
```

```
        logger.info("Received: " + user);
    }
}
```

- 为了将 Sink Receiver 的输入通道目标设置为greetings 主题，以及将该服务的实例设置为同一个消费组，可做如下设置：

```
spring.cloud.stream.bindings.input.group=Service-A
spring.cloud.stream.bindings.input.destination=greetings
```

通过 spring.cloud.stream.bindings.input.group 属性指定了该应用实例都属于 Service-A 消费组，而 spring.cloud.stream.bindings.input.destination 属性则指定了输入通道对应的主题名。

- 完成了消息消费者应用之后，我们再来实现一个消息生产者应用 Sink Sender,具体如下：

```
@EnableBinding(value = {Source.class})
public class SinkSender {

    private static Logger logger = LoggerFactory.getLogger(SinkSender.class);

    @Bean
    @InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay =
"2000"))
    public MessageSource<String> timerMessageSource() {
        return () -> new GenericMessage<>("{\"name\":\"didi\", \"age\":30}");
    }


}
```

- 为消息生产者 Sink Seder 做一些设置，让它的输出通道绑定目标也指向 greetings 主题，具体如下：

```
spring.cloud.stream.bindings.output.destination=greetings
```

到这里，对于消费分组的示例就完成了。分别运行上面实现的生产者与消费者，其中消费者我们启动多个实例。通过控制台，可以发现，每个生产者发出的消息会被启动的消费者以轮询的方式进行接收和输出。

消息分区

通过消费组的设置，虽然我们已经能够在多实例环境下，保证同一消息只被一个消费者实例进行接收和处理，但是，对于一些特殊场景，除了要保证单一实例消费之外，还希望那些具备相同特征的消息都能够被同一个实例进行消费。这时候我们就需要对消息进行分区处理。

在 Spring Cloud Stream 中实现消息分区非常简单，我们对消费组示例做一些配置修改就能实现，具体如下所示。

- 在消费者应用 SinkReceiver 中，对配置文件做一些修改，具体如下：

```
spring.cloud.stream.bindings.input.group=Service-A

spring.cloud.stream.bindings.input.destination=greetings
spring.cloud.stream.bindings.input.consumer.partitioned=true
spring.cloud.stream.instanceCount=2
spring.cloud.stream.instanceIndex=0
```

从上面的配置中，我们可以看到增加了下面这三个参数。

1. spring.cloud.stream.bindings.input.consumer.partitioned：通过该参数开启消费者分区功能。
2. spring.cloud.stream.instanceCount：该参数指定了当前消费者的总实例数量。
3. spring.cloud.stream.instanceIndex：该参数设置当前实例的索引号，从 0 开始，最大值为 spring.cloud.stream.instanceCount 参数-1。试验的时候需要启动多个实例，可以通过运行参数来为不同实例设置不同的索引值。

- 在生产者应用 SinkSender 中，对配置文件也做一些修改，具体如下所示。

```
spring.cloud.stream.bindings.output.destination=greetings
spring.cloud.stream.bindings.output.producer.partitionKeyExpression=payload
spring.cloud.stream.bindings.output.producer.partitionCount=2
```

从上面的配置中，我们可以看到增加了下面这两个参数。

1. spring.cloud.stream.bindings.output.producer.partitionKey-Expression：通过该参数指定了分区键的表达式规则，我们可以根据实际的输出消息规则配置 SpEL 来生成合适的分区键。
2. spring.cloud.stream.bindings.output.producer.partitionCount：该参数指定了消息分区的数量。

到这里消息分区配置就完成了，我们可以再次启动这两个应用，同时启动多个消费者。但需要注意的是，要为消费者指定不同的实例索引号，这样当同一个消息被发送给消费组时，可以发现只有一个消费实例在接收和处理这些相同的消息。

消息类型

Spring Cloud Stream 为了让开发者能够在消息中声明它的内容类型，在输出消息中定

义了一个默认的头信息：contentType。对于那些不直接支持头信息的消息中间件，Spring Clou d Stream 提供了自己的实现机制，它会在消息发出前自动将消息包装进它自定义的消息封装格式中，并加入头信息。而对于那些自身就支持头信息的消息中间件，Spring Clou d Stream 构建的服务可以接收并处理来自非 Spring Clou dStream 构建但包含符合规范头信息的应用程序发出的消息。

Spring Clou dStream 允许使用 spring.cloud.stream.bindngs.<channelName>.content-type 属性以声明式的配置方式为绑定的输入和输出通道设置消息内容的类型。此外，原生的消息类型转换器依然可以轻松地用于我们的应用程序。目前，Spring Clou d Stream 中自带支持了以下几种常用的消息类型转换。

- JSON 与 POJO 的互相转换。
- JSON 与 org.springframework.tuple.Tuple 的互相转换。
- Object 与 byte[]的互相转换。为了实现远程传输序列化的原始字节，应用程序需要发送 byte 类型的数据，或是通过实现 Jav a 的序列化接口来转换为字节（Object 对象必须可序列化）。
- String 与 byte[]的互相转换。
- Object 向纯文本的转换：Object 需要实现 toString()方法。

上面所指的 JSON 类型可以表现为一个 byte 类型的数组，也可以是一个包含有效 JSON 内容的字符串。另外，Object 对象可以由 JSON、byte 数组或者字符串转换而来，但是在转换为 JSON 的时候总是以字符串的形式返回。

MIME 类型

在 Spring Clou dStream 中定义的 content-type 属性采用了 Media Type，即 Internet Media Ty pe（互联网媒体类型），也被称为 MIME 类型，常见的有 application/json、text/plain;charset=UTF-8，相信接触过 HTTP 的工程师们对这些类型都不会感到陌生。MIME 类型对于标示如何转换为 String 或 byte[]非常有用。并且，我们还可以使用 MIME 类型格式来表示 Jav a类型，只需要使用带有类型参数的一般类型：application/x-java-object。比如，我们可以使用 application/x-java-object;type=java.util.Map 来表示传输的是一个 java.util.Map 对象，或是使用 application/x-java-object;type=com.didispace.User 来表示传输的是一个 com.didispace.User 对象；除此之外，更重要的是，它还提供了自定义的 MIME 类型，比如通过 application/x-spring-tuple 来指定 Spring 的 Tuple 类型。

在 Spring Clou d Stream 中默认提供了一些可以开箱即用的类型转换器，具体如下表所示。

源内容类型	目标内容类型	content-type 头	content-type	注释
POJO	JSON String	ignored	application/json	
Tuple	JSON String	ignored	application/json	JSON is tailored for Tuple
POJO	String (toString())	ignored	text/plain, java.lang.String	
POJO	byte[] (java.io serialized)	ignored	application/x-java-serialized-object	
JSON byte[] 或 String	POJO	application/json (or none)	application/x-java-object	
byte[]或 String	Serializable	application/x-java-serialized-object	application/x-java-object	
JSON byte[] 或 String	Tuple	application/json (or none)	application/x-spring-tuple	
byte[]	String	any	text/plain, java.lang.String	will apply any Charset specified in the content-type header
String	byte[]	any	application/octet-stream	will apply any Charset specified in the content-type header

消息类型的转换行为只会在需要进行转换时才被执行，比如，当服务模块产生了一个头信息为`application/json`的 XML 字符串消息，Spring Cloud Stream 是不会将该 XML 字符串转换为 JSON 的，这是因为该模块的输出内容已经是一个字符串类型了，所以它并不会将其做进一步的转换。

另外需要注意的是，Spring Cloud Stream 虽然同时支持输入通道和输出通道的消息类型转换，但还是推荐开发者尽量在输出通道中做消息转换。因为对于输入通道的消费者来说，当目标是一个 POJO 的时候，使用`@StreamListener`注解是能够支持自动对其进行转换的。

Spring Cloud Stream 除了提供上面这些开箱即用的转换器之外，还支持开发者自定义的消息转换器。这使得我们可以使用任意格式（包括二进制）的数据进行发送和接收，并且将这些数据与特定的`contentType`相关联。在应用启用的时候，Spring Cloud Stream 会将所有`org.springframework.messaging.converter.MessageConverter`接口实现的自定义转换器以及默认实现的那些转换器都加载到消息转换工厂中，以提供给消息处理时使用。

绑定器详解

在“核心概念”一节中，我们已经简单介绍过 Binder 绑定器的基本概念和作用：它是定义在应用程序与消息中间件之间的抽象层，用来屏蔽消息中间件对应用的复杂性，并提供简单而统一的操作接口给应用程序使用。在本节中，我们将详细介绍绑定器背后的细节和行为。

绑定器 SPI

绑定器 SPI 涵盖了一套可插拔的用于连接外部中间件的实现机制，其中包含了许多接口、开箱即用的实现类以及发现策略等内容。其中，最为关键的就是 Binder 接口，它是用来将输入和输出连接到外部中间件的抽象：

```
public interface Binder<T, C extends ConsumerProperties, P extends
ProducerProperties> {

    Binding<T> bindConsumer(String name, String group, T inboundBindTarget, C
consumerProperties);

    Binding<T> bindProducer(String name, T outboundBindTarget, P
producerProperties);

}
```

当应用程序对输入和输出通道进行绑定的时候，实际上就是通过该接口的实现来完成的。

- 向消息通道发送数据的生产者调用 bindProducer 方法来绑定输出通道时，第一个参数代表了发往消息中间件的目标名称，第二个参数代表了发送消息的本地通道实例，第三个参数是用来创建通道时使用的属性配置（比如分区键的表达式等）。
- 从消息通道接收数据的消费者调用 bindConsumer 方法来绑定输入通道时，第一个参数代表了接收消息中间件的目标名称，第二个参数代表了消费组的名称（如果多个消费者实例使用相同的组名，则消息将对这些消费者实例实现负载均衡，每个生产者发出的消息只会被组内一个消费者实例接收和处理），第三个参数代表了接收消息的本地通道实例，第四个参数是用来创建通道时使用的属性配置。

另外，从 Binder 的定义中，我们还可以知道 Binder 是一个参数化并且可扩展的接口。

- 对于输入与输出的绑定类型，在 1.0 版本中仅支持 MessageChannel，但是在接口中通过泛型定义，所以在未来可以对其进行扩展。
- 对于属性配置也提供了可扩展的定义，我们可以为特定的 Binder 以类型安全的方式来补充一些特有的属性。

一个典型的 Binder 绑定器实现一般包含以下内容。

- 一个实现 Binder 接口的类。
- 一个 Spring 配置加载类，用来创建连接消息中间件的基础结构使用的实例。
- 一个或多个能够在 classpath 下的 META-INF/spring.binders 路径找到的绑定器定义文件。比如我们可以在 spring-cloud-starter-stream-rabbit 中找到该文件，该文件中存储了当前绑定器要使用的自动化配置类的路径：

```
rabbit:\
org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfiguration
```

自动化配置

Spring Cloud Stream 通过绑定器 SPI 的实现将应用程序逻辑上的输入输出通道连接到物理上的消息中间件。消息中间件之间通常都会有或多或少的差异性，所以为了适配不同的消息中间件，需要为它们实现各自独有的绑定器。目前，Spring Cloud Stream 中默认实现了对 RabbitMQ、Kafka 的绑定器，在上面的示例中我们引入的 spring-cloud-starter-stream-rabbit 依赖中就包含了 RabbitMQ 的绑定器 spring-cloud-stream-binder-rabbit。

默认情况下，Spring Cloud Stream 也遵循 Spring Boot 自动化配置的特性。如果在 classpath 中能够找到单个绑定器的实现，那么 Spring Cloud Stream 会自动加载它。而我们在 classpath 中引入绑定器的方法也非常简单，只需要在 pom.xml 中增加对应消息中间件的绑定器依赖即可，比如：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

如果使用 Kafka，则引入：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
```

多绑定器配置

当应用程序的 classpath 下存在多个绑定器时，Spring Cloud Stream 在为消息通道做绑定操作时，无法判断应该使用哪个具体的绑定器，所以我们需要为每个输入或输出通道指定具体的绑定器。

我们在一个应用程序中使用多个绑定器时，往往其中一个绑定器会是主要使用的，而第二个可能是为了适应一些特殊要求（比如性能等原因）。我们可以先通过设置默认绑定器来为大部分的通道设置绑定器。比如，使用 RabbitMQ 设置默认绑定器：

```
spring.cloud.stream.defaultBinder=rabbit
```

在设置了默认绑定器之后，再为其他一些少数的消息通道单独设置绑定器，比如：

```
spring.cloud.stream.bindings.input.binder=kafka
```

需要注意的是，上面我们设置参数时用来指定具体绑定器的值并不是消息中间件的名称，而是在每个绑定器实现的 META-INF/spring.binders 文件中定义的标识（一个绑定器实现的标识可以定义多个，以逗号分隔），所以上面配置的 rabbit 和 kafka 分别来自于各自的配置定义，它们的具体内容如下所示：

```
    rabbit:\
    org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfigur
ation

    kafka:\
    org.springframework.cloud.stream.binder.kafka.config.KafkaBinderConfiguration
```

另外，当需要在一个应用程序中使用同一类型不同环境的绑定器时，我们也可以通过配置轻松实现通道绑定。比如，当需要连接两个不同的 RabbitMQ 实例的时候，可以参照如下配置：

```
spring.cloud.stream.bindings.input.binder=rabbit1
spring.cloud.stream.bindings.output.binder=rabbit2

spring.cloud.stream.binders.rabbit1.type=rabbit
spring.cloud.stream.binders.rabbit1.environment.spring.rabbitmq.host=192.168.0.101
spring.cloud.stream.binders.rabbit1.environment.spring.rabbitmq.port=5672
spring.cloud.stream.binders.rabbit1.environment.spring.rabbitmq.username=springcloud
spring.cloud.stream.binders.rabbit1.environment.spring.rabbitmq.password=123456

spring.cloud.stream.binders.rabbit2.type=rabbit
spring.cloud.stream.binders.rabbit2.environment.spring.rabbitmq.host=192.168.0.102
spring.cloud.stream.binders.rabbit2.environment.spring.rabbitmq.port=5672
spring.cloud.stream.binders.rabbit2.environment.spring.rabbitmq.username=springcloud
spring.cloud.stream.binders.rabbit2.environment.spring.rabbitmq.password=123456
```

从上面的配置中，我们可以看到对于输入输出通道指定的绑定器采用了显式别名的配置方式，其中 input 通道的绑定指定了名为 rabbit1 的配置，而 output 通道的绑定指定了名为 rabbit2 的配置。当采用显式配置方式时会自动禁用默认的绑定器配置，所以当定义了显式配置别名后，对于这些绑定器的配置需要通过 spring.cloud.stream.

binders.<configurationName>属性来进行设置。对于绑定器的配置主要有下面 4 个参数。

- spring.cloud.stream.binders.<configurationName>.type 指定了绑定器的类型，可以是 rabbit、kafak 或者其他自定义绑定器的标识名，绑定器标识名的定义位于绑定器的 META-INF/spring.binders 文件中。
- spring.cloud.stream.binders.<configurationName>.environment 参数可以直接用来设置各绑定器的属性，默认为空。
- spring.cloud.stream.binders.<configurationName>.inheritEnvironment 参数用来配置当前绑定器是否继承应用程序自身的环境配置，默认为 true。
- spring.cloud.stream.binders.<configurationName>.defaultCandidate 参数用来设置当前绑定器配置是否被视为默认绑定器的候选项，默认为 true。当需要让当前配置不影响默认配置时，可以将该属性设置为 false。

RabbitMQ 与 Kafka 绑定器

在之前的章节中我们多次提到，Spring Cloud Stream 自身就提供了对 RabbitMQ 和 Kafka 的绑定器实现。由于 RabbitMQ 和 Kafka 自身的实现结构有所不同，理解绑定器实现与消息中间件自有概念之间的对应关系，对于正确使用绑定器和消息中间件会有非常大的帮助。下面就来分别说说 RabbitMQ 与 Kafka 的绑定器是如何使用消息中间件中不同概念来实现消息的生产与消费的。

- RabbitMQ 绑定器：在 RabbitMQ 中，通过 Exchange 交换器来实现 Spring Cloud Stream 的主题概念，所以消息通道的输入输出目标映射了一个具体的 Exchange 交换器。而对于每个消费组，则会为对应的 Exchange 交换器绑定一个 Queue 队列进行消息收发。
- Kafka 绑定器：由于 Kafka 自身就有 Topic 概念，所以 Spring Cloud Stream 的主题直接采用了 Kafka 的 Topic 主题概念，每个消费组的通道目标都会直接连接 Kafka 的主题进行消息收发。

配置详解

在 Spring Cloud Stream 中对绑定通道和绑定器提供了通用的属性配置项，一些绑定器还允许使用附加属性来对消息中间件的一些独有特性进行配置。这些属性的配置可以通过 Spring Boot 支持的任何配置方式来进行，包括使用环境变量、YAML 或者 properties 配置文件等。

基础配置

下表是Spring Cloud Stream应用级别的通用基础属性，这些属性都以spring.cloud.stream.为前缀。

参数名	说明	默认值
instanceCount	应用程序部署的实例数量。当使用 Kafka 的时候需要设置分区	1
instanceIndex	应用程序实例的索引，该值从 0 开始，最大值设置为-1。当使用分区和 Kafka 的时候使用	
dynamicDestinations	动态绑定的目标列表，该列表默认为空，当设置了具体列表之后，只有列表中的目标才能被发现	空
defaultBinder	默认绑定器配置，在应用程序中有多个绑定器时使用	空
overrideCloudConnectors	该属性只适用于激活cloud配置并且提供了Spring Cloud Connectors的应用。当使用默认属性false时，绑定器会自动检测合适的服务来绑定（比如，在 Cloud Foundry 中绑定的 RabbitMQ 服务）。当设置为true时，绑定器将忽略绑定的服务，而是依赖应用程序中的设置属性来进行绑定和连接	false

绑定通道配置

对于绑定通道的属性配置，在之前的示例中已经有过一些介绍，这些配置可以在属性文件中通过 spring.cloud.stream.bindings.<channelName>.<property>=<value>格式的参数来进行设置。其中<channelName>代表在绑定接口中定义的通道名称，比如，Sink 中的 input、Source 中的 output。

由于绑定通道分为输入通道和输出通道，所以在绑定通道的配置中包含了三类面向不同通道类型的配置：通用配置、消费者配置、生产者配置。

在下面介绍各具体配置属性时将省略 spring.cloud.stream.bindings.<channelName>.前缀，但在实际使用的时候记得使用完整的参数名称进行配置。

通用配置

对于绑定通道的通用配置，它们既适用于输入通道，也适用于输出通道，它们通过spring.cloud.stream.bindings.<channelName>.前缀来进行设置，具体可配置的属性如下表所示。

参数名	说明	默认值
destination	该参数用来配置消息通道绑定在消息中间件中的目标名称，比如 RabbitMQ 的 Exchange 或 Kafka 的 Topic。如果配置的绑定通道是一个消费者（输入通道），那么它可以绑定多个目标，这些目标名称通过逗号来分隔。如果没有设置该属性，将使用通道名	
group	该参数用来设置绑定通道的消费组，该参数主要作用于输入通道，以保证同一消息组中的消息只会有一个消费示例接收和处理	null
contentType	该参数用来设置绑定通道的消息类型	null
binder	当存在多个绑定器时使用该参数来指定当前通道使用哪个具体的绑定器	null

消费者配置

下面这些配置仅对输入通道的绑定有效，它们以 spring.cloud.stream.bindings.<channelName>.consumer.格式作为前缀。

参数名	说明	默认值
concurrency	输入通道消费者的并发数	1
partitioned	来自消息生产者的数据是否采用了分区	false
headerMode	当设置为 raw 的时候将禁用对消息头的解析。该属性只有在使用不支持消息头功能的中间件时有效，因为 Spring Cloud Stream 默认会解析嵌入的头部信息	embeddedHeaders
maxAttempts	对输入通道消息处理的最大重试次数	3
backOffInitialInterval	重试消息处理的初始间隔时间	1000
backOffMaxInterval	重试消息处理的最大间隔时间	10000
backOffMultiplier	重试消息处理时间间隔的递增乘数	2.0

生产者配置

下面这些配置仅对输出通道的绑定有效，它们以 spring.cloud.stream.bindings.<channelName>.producer.格式作为前缀。

参数名	说明	默认值
partitionKeyExpression	该参数用来配置输出通道数据分区键的 SpEL 表达式。当设置该属性之后，将对当前绑定通道的输出数据进行分区处理。同时，partitionCount 参数必须大于 1 才能生效。该参数与 partitionKeyExtractorClass 参数互斥，不能同时设置	null

续表

参数名	说明	默认值
partitionKeyExtractorClass	该参数用来配置分区键提取策略接口 PartitionKeyExtractorStrategy 的实现。当设置该属性之后，将对当前绑定通道的输出数据进行分区处理。同时，partitionCount 参数必须大于 1 才能生效。该参数与 partitionKeyExpression 参数互斥，不能同时设置	null
partitionSelectorClass	该参数用来指定分区选择器接口 PartitionSelectorStrategy 的实现。它与 partitionSelectorExpression 参数互斥，不能同时设置。如果两者都不设置，那么分区选择计算规则为 hashCode(key)% partitionCount，这里的 key 根据 partitionKeyExpression 或 partition-KeyExtractorClass 的配置计算得到	null
partitionSelectorExpression	该参数用来配置自定义分区选择器的 SpEL 表达式。它与 partitionSelectorClass 参数互斥，不能同时设置。如果两者都不设置，那么分区选择计算规则为 hashCode(key)% partitionCount，这里的 key 根据 partitionKeyExpression 或 partition-KeyExtractorClass 的配置计算得到	null
partitionCount	当分区功能开启时，使用该参数来配置消息数据的分区数。如果消息生产者已经配置了分区键的生成策略，那么它的值必须大于 1	1
headerMode	当设置为 raw 的时候将在输出时禁用嵌入式头部信息。该属性只有在使用不支持消息头功能的中间件时有效，因为 Spring Cloud Stream 会默认地为其嵌入头部信息	embeddedHeaders

绑定器配置

由于 Spring Cloud Stream 目前只实现了对 RabbitMQ 和 Kafka 的绑定器，所以对于绑定器的属性配置主要是针对这两个中间件。同时因为这两个中间件自身结构的不同，所以会

有不同的附加属性来配置各自的一些特殊功能和基础设施。

RabbitMQ 配置

RabbitMQ 绑定器的配置同绑定通道的配置一样，分为三种不同类型：通用配置、消费者配置以及生产者配置。

通用配置

由于 RabbitMQ 绑定器默认使用了 Spring Boot 的 ConnectionFactory，所以 RabbitMQ 绑定器支持在 Spring Boot 中的配置选项，它们以 spring.rabbitmq.为前缀。在之前的示例中，我们使用这些配置来指定具体的 RabbitMQ 地址、端口、用户信息等，更多配置可见 Spring Boot 文档中对 RabbitMQ 支持的章节内容，或是通过 spring-boot-starter-amqp 模块中 RabbitProperties 类的源码来查看。

在 Spring Cloud Stream 对 RabbitMQ 实现的绑定器中主要有下面几个属性，它们都以 spring.cloud.stream.rabbit.binder.为前缀。这些属性可以在 org.springframework.cloud.stream.binder.rabbit.config.RabbitBinderConfigurationProperties 中找到它们。

参数名	说明	默认值
adminAddresses	该参数用来配置 RabbitMQ 管理插件的 URL，当需要配置多个时用逗号分隔。该参数只有在 nodes 参数包含多个时使用，并且这里配置的内容必须在 spring.rabbitmq.addresses 中存在	
nodes	该参数用来配置 RabbitMQ 的节点名称，当需要配置多个时用逗号分隔。在配置多个的情况下，可以用来定位队列所在的服务器地址。这里配置的内容必须在 spring.rabbitmq.addresses 中存在	
compressionLevel	绑定通道的压缩级别，它的具体可选值及含义可见 java.util.zip.Deflater 中的定义	1

消费者配置

下面这些配置仅对 RabbitMQ 输入通道的绑定有效，它们以 spring.cloud.stream.rabbit.bindings.<channelName>.consumer.格式作为前缀。

参数名	说明	默认值
acknowledgeMode	用来设置消息的确认模式，可选配置包含：NONE、MANUAL、AUTO	AUTO

续表

参数名	说明	默认值
autoBindDlq	用来设置是否自动声明 DLQ（Dead-Letter-Queue），并绑定到 DLX（Dead-Letter-Exchange）上	false
durableSubscription	用来设置订阅是否被持久化，该参数仅在 group 被设置的时候有效	true
maxConcurrency	用来设置消费者的最大并发数	1
prefetch	用来设置预取数量，它表示在一次会话中从消息中间件中获取的消息数量，该值越大消息处理越快，但是会导致非顺序处理的风险	1
prefix	用来设置统一的目标和队列名称前缀	
recoveryInterval	用来设置恢复连接的尝试时间间隔，以毫秒为单位	5000
requeueRejected	用来设置消息传递失败时重传	true
requestHeaderPatterns	用来设置需要被传递的请求头信息	[STANDARD_REQUEST_HEADERS,'*']
replyHeaderPatterns	用来设置需要被传递的响应头信息	[STANDARD_REPLY_HEADERS,'*']
republishToDlq	默认情况下，消息在重试也失败之后会被拒绝。如果 DLQ 被配置的时候，RabbitMQ 会将失败的消息路由到 DLQ 中。如果该参数设置为 true，总线会将失败的消息附加一些头信息（包括异常消息、引起失败的跟踪堆栈）之后重新发布到 DLQ 中	

续表

参数名	说明	默认值
transacted	用来设置是否启用 channel-transacted，即是否在消息中使用事务	false
txSize	用来设置 transaction-size 的数量，当 acknowledge-Mode 被设置为 AUTO 时，容器会在处理 txSize 数目消息之后才开始应答	1

生产者配置

下面这些配置仅对 RabbitMQ 输出通道的绑定有效，它们以 spring.cloud.stream.rabbit.bindings.<channelName>.producer.格式作为前缀。

参数名	说明	默认值
autoBindDlq	用来设置是否自动声明 DLQ（Dead-Letter-Queue），并绑定到 DLX（Dead-Letter-Exchange）上	false
batchingEnabled	是否启用消息批处理	false
batchSize	当批处理开启时，用来设置缓存的批处理消息数量	100
batchBufferLimit	批处理缓存限制	10000
batchTimeout	批处理超时时间	5000
compress	消息发送时是否启用压缩	false
deliveryMode	消息发送模式	PERSISTENT
prefix	用来设置统一的目标前缀	
requestHeaderPatterns	用来设置需要被传递的请求头信息	[STANDARD_REQUEST_HEADERS,'*']
replyHeaderPatterns	用来设置需要被传递的响应头信息	[STANDARD_REPLY_HEADERS,'*']

Kafka 配置

Kafka 绑定器的配置在类别上与 RabbitMQ 一样，分为三种不同类型：通用配置、消费

者配置以及生产者配置。但是由于 RabbitMQ 与 Kafka 自身有一些差异，所以它们的配置也不一样。

通用配置

Spring Cloud Stream 实现的 Kafka 绑定器包含下面这些通用配置，它们都以 spring.cloud.stream.kafka.binder.为前缀。

参数名	说明	默认值
brokers	Kafka 绑定器连接的消息中间件列表。需要配置多个时用逗号分隔，每个地址可以是单独的 host，也可以是 host:port 的形式	localhost
defaultBrokerPort	用来设置默认的消息中间件端口号。当 brokers 中的配置地址没有包含端口信息时，将使用该参数配置的默认端口进行连接	9092
zkNodes	Kafka 绑定器使用的 ZooKeeper 节点列表。需要配置多个时用逗号分隔，每个地址可以是单独的 host，也可以是 host:port 的形式	localhost
defaultZkPort	用来设置默认的 ZooKeeper 端口号。当 zkNodes 中的配置地址没有包含端口信息时，将使用该参数配置的默认端口进行连接	2181
headers	用来设置会被传输的自定义头信息	
offsetUpdateTimeWindow	用来设置 offset 的更新频率，以毫秒为单位，如果设置为 0 则忽略	10000
offsetUpdateCount	用来设置 offset 以次数表示的更新频率，如果为 0 则忽略，该参数与 offsetUpdateTimeWindow 互斥	0
requiredAcks	用来设置确认消息的数量	1
minPartitionCount	该参数仅在设置了 autoCreateTopics 和 autoAddPartitions 时生效，用来设置该绑定器所使用主题的全局分区最小数量。如果当生产者的 partitionCount 参数或 instanceCount * concurrency 设置大于该参数配置时，该参数值将被覆盖	1
replicationFactor	当 autoCreateTopics 参数为 true 时候，用来配置自动创建主题的副本数量	1
autoCreateTopics	该参数默认为 true，绑定器会自动地创建新主题。如果设置为 false，那么绑定器将使用已经配置的主题，但是在这种情况下，如果需要使用的主题不存在，绑定器会启动失败	true

续表

参数名	说明	默认值
autoAddPartitions	该参数默认为 false，绑定器会根据已经配置的主题分区来实现，如果目标主题的分区数小于预期值，那么绑定器会启动失败。如果该参数设置为 true，绑定器将在需要的时候自动创建新的分区	false
socketBufferSize	该参数用来设置 Kafka 的 Socket 缓存大小	2097152

消费者配置

下面这些配置仅对 Kafka 输入通道的绑定有效，它们以 spring.cloud.stream.kafka.bindings.<channelName>.consumer.格式作为前缀。

参数名	说明	默认值
autoCommitOffset	用来设置是否在处理消息时自动提交 offset。如果设置为 false，在消息头中会加入 ACK 头信息以实现延迟确认	true
autoCommitOnError	该参数只有在 autoCommitOffset 设置为 true 时才有效。当设置为 false 的时候，引起错误的消息不会自动提交 offset，仅提交成功消息的 offset。如果设置为 true，不论消息是否成功，都会自动提交。当不设置该值时，它实际上具有与 enableDlq 相同的配置值	
recoveryInterval	尝试恢复连接的时间间隔，以毫秒为单位	5000
resetOffsets	是否使用提供的 startOffset 值来重置消费者的 offset 值	false
startOffset	用来设置新建组的起始 offset，该值也会在 resetOffsets 开始时被使用	null
enableDlq	该参数设置为 true 时，将为消费者启用 DLQ 行为，引起错误的消息将被发送到名为 error.<destination>.<group>的主题中去	false

生产者配置

下面这些配置仅对 Kafka 输出通道的绑定有效，它们以 spring.cloud.stream.kafka.bindings.<channelName>.producer.格式作为前缀。

参数名	说明	默认值
bufferSize	Kafka 批量发送前的缓存数据上限，以字节为单位	16384
sync	该参数用来设置 Kafka 消息生产者的发送模式，默认为 false，即采用 async 配置，允许批量发送数据。当设置为 true 时，将采用 sync 配置，消息将不会被批量发送，而是一条一条地发送	false

续表

参数名	说明	默认值
batchTimeout	消息生产者批量发送时，为了积累更多发送数据而设置的等待时间。通常情况下，生产者基本不会等待，而是直接发送所有在前一批次发送时积累的消息数据。当我们设置一个非 0 值时，可以以延迟为代价来增加系统的吞吐量	0

第 11 章

分布式服务跟踪：Spring Cloud Sleuth

通过之前各章节介绍的 Spring Cloud 组件，实际上我们已经能够通过使用它们搭建起一个基础的微服务架构系统来实现业务需求了。但是，随着业务的发展，系统规模也会变得越来越大，各微服务间的调用关系也变得越来越错综复杂。通常一个由客户端发起的请求在后端系统中会经过多个不同的微服务调用来协同产生最后的请求结果，在复杂的微服务架构系统中，几乎每一个前端请求都会形成一条复杂的分布式服务调用链路，在每条链路中任何一个依赖服务出现延迟过高或错误的时候都有可能引起请求最后的失败。这时候，对于每个请求，全链路调用的跟踪就变得越来越重要，通过实现对请求调用的跟踪可以帮助我们快速发现错误根源以及监控分析每条请求链路上的性能瓶颈等。

针对上面所述的分布式服务跟踪问题，Spring Cloud Sleuth 提供了一套完整的解决方案。在本章中，我们将详细介绍如何使用 Spring Cloud Sleuth 来为微服务架构增加分布式服务跟踪的能力。

快速入门

在介绍各种概念与原理之前，我们先通过实现一个简单的示例，为存在服务调用的应用增加一些 Sleuth 的配置以实现基本的服务跟踪功能，以此来对 Spring Cloud Sleuth 有一个初步的了解，随后再逐步展开，介绍实现过程中的各个细节。

准备工作

在引入 Sleuth 之前，我们先按照之前章节学习的内容来做一些准备工作，构建一些基

础的设施和应用。

- 服务注册中心：eureka-server，这里不做赘述，直接使用之前构建的工程即可。
- 微服务应用：trace-1，实现一个 REST 接口/trace-1，调用该接口后将触发对 trace-2 应用的调用。具体实现如下所述。
    - 创建一个基础的 Spring Boot 应用，在 pom.xml 中增加下面的依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/>
</parent>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

    - 创建应用主类，实现/trace-1 接口，并使用 RestTemplate 调用 trace-2 应用的接口。具体如下：

```
@RestController
@EnableDiscoveryClient
@SpringBootApplication
public class TraceApplication {
```

```
    private final Logger logger = Logger.getLogger(getClass());

    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @RequestMapping(value = "/trace-1", method = RequestMethod.GET)
    public String trace() {
        logger.info("===call trace-1===");
        return restTemplate().getForEntity("http://trace-2/trace-2",
String.class).getBody();
    }

    public static void main(String[] args) {
        SpringApplication.run(TraceApplication.class, args);
    }

}
```

- 在 application.properties 中，将 eureka.client.serviceUrl.defaultZone 参数指向 eureka-server 的地址，具体如下：

```
spring.application.name=trace-1
server.port=9101
eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

- 微服务应用：trace-2，实现一个 REST 接口/trace-2，供 trace-1 调用。具体实现如下所示。
  - 创建一个基础的 Spring Boot 应用，pom.xml 中的依赖与 trace-1 相同。
  - 创建应用主类，并实现/trace-2 接口，具体实现如下：

```
@RestController
@EnableDiscoveryClient
@SpringBootApplication
public class TraceApplication {

    private final Logger logger = Logger.getLogger(getClass());

    @RequestMapping(value = "/trace-2", method = RequestMethod.GET)
    public String trace() {
        logger.info("===<call trace-2>===");
        return "Trace";
    }
```

```
    public static void main(String[] args) {
        SpringApplication.run(TraceApplication.class, args);
    }

}
```

- 在application.properties中，将eureka.client.serviceUrl.defaultZone参数指向 eureka-server 的地址，另外还需要设置不同的应用名和端口号，具体如下：

```
spring.application.name=trace-2
server.port=9102
eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

在实现了上面的内容之后，我们可以将 eureka-server、trace-1、trace-2 三个应用都启动起来，并通过 postman 或 curl 等工具来对 trace-1 的接口发送请求 http://localhost:9101/trace-1。可以得到返回值 Trace，同时还能在它们的控制台中分别获得下面的输出：

```
-- trace-1
INFO 25272 --- [nio-9101-exec-2] ication$$EnhancerBySpringCGLIB$$36e12c68 :
===<call trace-1>===

-- trace-2
INFO 7136 --- [nio-9102-exec-1] ication$$EnhancerBySpringCGLIB$$52a02f0b : ===<call
trace-2>===
```

实现跟踪

在完成了准备工作之后，接下来我们开始进行本章的主题内容，为上面的 trace-1 和 trace-2 添加服务跟踪功能。通过 Spring Cloud Sleuth 的封装，我们为应用增加服务跟踪能力的操作非常简单，只需在 trace-1 和 trace-2 的 pom.xml 依赖管理中增加 spring-cloud-starter-sleuth 依赖即可，具体如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

到这里，实际上我们已经为 trace-1 和 trace-2 实现服务跟踪做好了基础的准备，重启 trace-1 和 trace-2，再对 trace-1 的接口发送请求 http://localhost:9101/trace-1。此时，我们可以从它们的控制台输出中窥探到 Sleuth 的一些端倪。

```
-- trace-1
INFO [trace-1,f410ab57afd5c145,a9f2118fa2019684,false] 25028 --- [nio-9101-exec-1]
```

```
ication$$EnhancerBySpringCGLIB$$d8228493 : ===<call trace-1>===

    -- trace-2
    INFO [trace-2,f410ab57afd5c145,e9a377dc2268bc29,false] 23112 --- [nio-9102-exec-1]
ication$$EnhancerBySpringCGLIB$$e6cb4078 : ===<call trace-2>===
```

从上面的控制台输出内容中，我们可以看到多了一些形如[trace-1, f410ab57afd5c145,a9f2118fa2019684,false]的日志信息，而这些元素正是实现分布式服务跟踪的重要组成部分，每个值的含义如下所述。

- 第一个值：trace-1，它记录了应用的名称，也就是 application.properties 中 spring.application.name 参数配置的属性。
- 第二个值：f410ab57afd5c145，Spring Cloud Sleuth 生成的一个 ID，称为 Trace ID，它用来标识一条请求链路。一条请求链路中包含一个 Trace ID，多个 Span ID。
- 第三个值：a9f2118fa2019684，Spring Cloud Sleuth 生成的另外一个 ID，称为 Span ID，它表示一个基本的工作单元，比如发送一个 HTTP 请求。
- 第四个值：false，表示是否要将该信息输出到 Zipkin 等服务中来收集和展示。

上面四个值中的 Trace ID 和 Span ID 是 Spring Cloud Sleuth 实现分布式服务跟踪的核心。在一次服务请求链路的调用过程中，会保持并传递同一个 Trace ID，从而将整个分布于不同微服务进程中的请求跟踪信息串联起来。以上面输出内容为例，trace-1 和 trace-2 同属于一个前端服务请求来源，所以它们的 Trace ID 是相同的，处于同一条请求链路中。

跟踪原理

分布式系统中的服务跟踪在理论上并不复杂，它主要包括下面两个关键点。

- 为了实现请求跟踪，当请求发送到分布式系统的入口端点时，只需要服务跟踪框架为该请求创建一个唯一的跟踪标识，同时在分布式系统内部流转的时候，框架始终保持传递该唯一标识，直到返回给请求方为止，这个唯一标识就是前文中提到的 Trace ID。通过 Trace ID 的记录，我们就能将所有请求过程的日志关联起来。
- 为了统计各处理单元的时间延迟，当请求到达各个服务组件时，或是处理逻辑到达某个状态时，也通过一个唯一标识来标记它的开始、具体过程以及结束，该标识就是前文中提到的 Span ID。对于每个 Span 来说，它必须有开始和结束两个节点，通过记录开始 Span 和结束 Span 的时间戳，就能统计出该 Span 的时间延迟，除了时间戳记录之外，它还可以包含一些其他元数据，比如事件名称、请求信息等。

在快速入门示例中，我们轻松实现了日志级别的跟踪信息接入，这完全归功于

spring-cloud-starter-sleuth 组件的实现。在 Spring Boot 应用中，通过在工程中引入 spring-cloud-starter-sleuth 依赖之后，它会自动为当前应用构建起各通信通道的跟踪机制，比如：

- 通过诸如 RabbitMQ、Kafka（或者其他任何 Spring Cloud Stream 绑定器实现的消息中间件）传递的请求。
- 通过 Zuul 代理传递的请求。
- 通过 RestTemplate 发起的请求。

在快速入门示例中，由于 trace-1 对 trace-2 发起的请求是通过 RestTemplate 实现的，所以 spring-cloud-starter-sleuth 组件会对该请求进行处理。在发送到 trace-2 之前，Sleuth 会在该请求的 Header 中增加实现跟踪需要的重要信息，主要有下面这几个（更多关于头信息的定义可以通过查看 org.springframework.cloud.sleuth.Span 的源码获取）。

- X-B3-TraceId：一条请求链路（Trace）的唯一标识，必需的值。
- X-B3-SpanId：一个工作单元（Span）的唯一标识，必需的值。
- X-B3-ParentSpanId：标识当前工作单元所属的上一个工作单元，Root Span（请求链路的第一个工作单元）的该值为空。
- X-B3-Sampled：是否被抽样输出的标志，1 表示需要被输出，0 表示不需要被输出。
- X-Span-Name：工作单元的名称。

可以通过对 trace-2 的实现做一些修改来输出这些头部信息，具体如下：

```
@RequestMapping(value = "/trace-2", method = RequestMethod.GET)
public String trace(HttpServletRequest request) {
   logger.info("===<call trace-2, TraceId={}, SpanId={}>===",
         request.getHeader("X-B3-TraceId"), request.getHeader("X-B3-SpanId"));
   return "Trace";
}
```

通过上面的改造，我们再运行快速入门的示例内容，并发起对 trace-1 的接口访问，可以得到如下输出内容。其中在 trace-2 的控制台中，输出了当前正在处理的 TraceID 和 SpanId 信息。

```
    -- trace-1
    INFO [trace-1,a6e9175ffd5d2c88,8524f519b8a9e399,true] 10532 --- [nio-9101-exec-2]
icationEnhancerBySpringCGLIB27aa9624 : ===<call trace-1>===

    -- trace-2
    INFO [trace-2,a6e9175ffd5d2c88,ce60dcf1e2ed918f,true] 1208 --- [nio-9102-exec-3]
icationEnhancerBySpringCGLIBa7d84797 : ===<call trace-2, TraceId=a6e9175ffd5d2c88,
SpanId=be4949ec115e554e>===
```

为了更直观地观察跟踪信息，我们还可以在 application.properties 中增加下面的配置：

```
logging.level.org.springframework.web.servlet.DispatcherServlet=DEBUG
```

通过将 Spring MVC 的请求分发日志级别调整为 DEBUG 级别，我们可以看到更多跟踪信息：

```
    -- trace-1
    2016-11-27 09:26:52.663 DEBUG [trace-1,a6e9175ffd5d2c88,a6e9175ffd5d2c88,true]
10532 --- [nio-9101-exec-2] o.s.web.servlet.DispatcherServlet       :
DispatcherServlet with name 'dispatcherServlet' processing GET request for [/trace-1]
    2016-11-27 09:26:52.666 DEBUG [trace-1,a6e9175ffd5d2c88,a6e9175ffd5d2c88,true]
10532 --- [nio-9101-exec-2] o.s.web.servlet.DispatcherServlet       : Last-Modified
value for [/trace-1] is: -1
    2016-11-27 09:26:52.685 DEBUG [trace-1,a6e9175ffd5d2c88,8524f519b8a9e399,true]
10532 --- [nio-9101-exec-2] o.s.web.servlet.DispatcherServlet       : Null ModelAndView
returned to DispatcherServlet with name 'dispatcherServlet': assuming HandlerAdapter
completed request handling
    2016-11-27 09:26:52.685 DEBUG [trace-1,a6e9175ffd5d2c88,a6e9175ffd5d2c88,true]
10532 --- [nio-9101-exec-2] o.s.web.servlet.DispatcherServlet       : Successfully
completed request

    -- trace-2
    2016-11-27 09:26:52.673 DEBUG [trace-2,a6e9175ffd5d2c88,be4949ec115e554e,true]
1208 --- [nio-9102-exec-3] o.s.web.servlet.DispatcherServlet       : DispatcherServlet
with name 'dispatcherServlet' processing GET request for [/trace-2]
    2016-11-27 09:26:52.679 DEBUG [trace-2,a6e9175ffd5d2c88,be4949ec115e554e,true]
1208 --- [nio-9102-exec-3] o.s.web.servlet.DispatcherServlet       : Last-Modified
value for [/trace-2] is: -1
    2016-11-27 09:26:52.682 DEBUG [trace-2,a6e9175ffd5d2c88,ce60dcf1e2ed918f,true]
1208 --- [nio-9102-exec-3] o.s.web.servlet.DispatcherServlet       : Null ModelAndView
returned to DispatcherServlet with name 'dispatcherServlet': assuming HandlerAdapter
completed request handling
    2016-11-27 09:26:52.683 DEBUG [trace-2,a6e9175ffd5d2c88,be4949ec115e554e,true]
1208 --- [nio-9102-exec-3] o.s.web.servlet.DispatcherServlet       : Successfully
completed request
```

抽样收集

通过 Trace ID 和 Span ID 已经实现了对分布式系统中的请求跟踪，而记录的跟踪信息最终会被分析系统收集起来，并用来实现对分布式系统的监控和分析功能，比如，预警延迟过长的请求链路、查询请求链路的调用明细等。此时，我们在对接分析系统时就会碰到一个问题：分析系统在收集跟踪信息的时候，需要收集多少跟踪信息才合适呢？

理论上来说，我们收集的跟踪信息越多就可以越好地反映出系统的实际运行情况，并给出更精准的预警和分析。但是在高并发的分布式系统运行时，大量的请求调用会产生海

量的跟踪日志信息，如果收集过多的跟踪信息将会对整个分布式系统的性能造成一定的影响，同时保存大量的日志信息也需要不少的存储开销。所以，在 Sleuth 中采用了抽象收集的方式来为跟踪信息打上收集标记，也就是我们之前在日志信息中看到的第 4 个布尔类型的值，它代表了该信息是否要被后续的跟踪信息收集器获取和存储。

Sleuth 中的抽样收集策略是通过 `Sampler` 接口实现的，它的定义如下：

```
public interface Sampler {
    /**
     * @return true if the span is not null and should be exported to the tracing
system
     */
    boolean isSampled(Span span);
}
```

通过实现 `isSampled` 方法，Spring Cloud Sleuth 会在产生跟踪信息的时候调用它来为跟踪信息生成是否要被收集的标志。需要注意的是，即使 `isSampled` 返回了 `false`，它仅代表该跟踪信息不被输出到后续对接的远程分析系统（比如 Zipkin），对于请求的跟踪活动依然会进行，所以我们在日志中还是能看到收集标识为 `false` 的记录。

默认情况下，Sleuth 会使用 `PercentageBasedSampler` 实现的抽样策略，以请求百分比的方式配置和收集跟踪信息。我们可以通过在 `application.properties` 中配置下面的参数对其百分比值进行设置，它的默认值为 `0.1`，代表收集 10%的请求跟踪信息。

```
spring.sleuth.sampler.percentage=0.1
```

在开发调试期间，通常会收集全部跟踪信息并输出到远程仓库，我们可以将其值设置为 `1`，或者也可以通过创建 `AlwaysSampler` 的 Bean（它实现的 `isSampled` 方法始终返回 `true`）来覆盖默认的 `PercentageBasedSampler` 策略，比如：

```
@Bean
public AlwaysSampler defaultSampler() {
    return new AlwaysSampler();
}
```

在实际使用时，通过与 Span 对象中存储信息的配合，我们可以根据实际情况做出更贴近需求的抽样策略，比如实现一个仅包含指定 Tag 的抽样策略：

```
public class TagSampler implements Sampler {

    private String tag;

    public TagSampler(String tag) {
        this.tag = tag;
    }
```

```
    @Override
    public boolean isSampled(Span span) {
        return span.tags().get(tag) != null;
    }
}
```

由于跟踪日志信息数据的价值往往仅在最近的一段时间内非常有用，比如一周。那么我们在设计抽样策略时，主要考虑在不对系统造成明显性能影响的情况下，以在日志保留时间窗内充分利用存储空间的原则来实现抽样策略。

与 Logstash 整合

通过之前的准备与整合，我们已经为 `trace-1` 和 `trace-2` 引入了 Spring Cloud Sleuth 的基础模块 `spring-cloud-starter-sleuth`，实现了在各个微服务的日志信息中添加跟踪信息的功能。但是，由于日志文件都离散地存储在各个服务实例的文件系统之上，仅仅通过查看日志文件来分析我们的请求链路依然是一件相当麻烦的事，所以我们还需要一些工具来帮助集中收集、存储和搜索这些跟踪信息。引入基于日志的分析系统是一个不错的选择，比如 ELK 平台，它可以轻松地帮助我们收集和存储这些跟踪日志，同时在需要的时候我们也可以根据 Trace ID 来轻松地搜索出对应请求链路相关的明细日志。

ELK 平台主要由 ElasticSearch、Logstash 和 Kibana 三个开源工具组成。

- ElasticSearch 是一个开源分布式搜索引擎，它的特点有：分布式，零配置，自动发现，索引自动分片，索引副本机制，RESTful 风格接口，多数据源，自动搜索负载等。
- Logstash 是一个完全开源的工具，它可以对日志进行收集、过滤，并将其存储供以后使用。
- Kibana 也是一个开源和免费的工具，它可以为 Logstash 和 ElasticSearch 提供日志分析友好的 Web 界面，可以帮助汇总、分析和搜索重要数据日志。

Spring Cloud Sleuth 在与 ELK 平台整合使用时，实际上只要实现与负责日志收集的 Logstash 完成数据对接即可，所以我们需要为 Logstash 准备 JSON 格式的日志输出。由于 Spring Boot 应用默认使用 logback 来记录日志，而 Logstash 自身也有对 logback 日志工具的支持工具，所以我们可以直接通过在 logback 的配置中增加对 Logstash 的 Appender，就能非常方便地将日志转换成以 JSON 的格式存储和输出了。

下面我们来详细介绍一下在快速入门示例的基础上，如何实现面向 Logstash 的日志输出配置。

- 在 `pom.xml` 依赖中引入 `logstash-logback-encoder` 依赖，具体如下：

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>4.6</version>
</dependency>
```

- 在工程/resource 目录下创建 bootstrap.properties 配置文件，将 spring.application.name=trace-1 配置移动到该文件中去。由于 logback-spring.xml 的加载在 application.properties 之前，所以之前的配置 logback-spring.xml 无法获取 spring.application.name 属性，因此这里将该属性移动到最先加载的 bootstrap.properties 配置文件中。
- 在工程/resource 目录下创建 logback 配置文件 logback-spring.xml，具体内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <include resource="org/springframework/boot/logging/logback/defaults.xml"/>

  <springProperty scope="context" name="springAppName" source="spring.
application.name"/>
  <!-- 日志在工程中的输出位置 -->
  <property name="LOG_FILE" value="${BUILD_FOLDER:-build}/${springAppName}"/>
<!-- 控制台的日志输出样式 -->
  <property name="CONSOLE_LOG_PATTERN"
          value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL_PATTERN:
-%5p}) %clr([${springAppName:-},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Expo
rt:-}]){yellow} %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %c
lr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wE
x}"/>

  <!-- 控制台 Appender -->
  <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
     <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
        <level>INFO</level>
     </filter>
     <encoder>
        <pattern>${CONSOLE_LOG_PATTERN}</pattern>
        <charset>utf8</charset>
     </encoder>
  </appender>

  <!-- 为 logstash 输出的 JSON 格式的 Appender -->
  <appender name="logstash" class="ch.qos.logback.core.rolling.RollingFileAppender">
```

```
        <file>${LOG_FILE}.json</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${LOG_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
            <providers>
                <timestamp>
                    <timeZone>UTC</timeZone>
                </timestamp>
                <pattern>
                    <pattern>
                        {
                        "severity": "%level",
                        "service": "${springAppName:-}",
                        "trace": "%X{X-B3-TraceId:-}",
                        "span": "%X{X-B3-SpanId:-}",
                        "exportable": "%X{X-Span-Export:-}",
                        "pid": "${PID:-}",
                        "thread": "%thread",
                        "class": "%logger{40}",
                        "rest": "%message"
                        }
                    </pattern>
                </pattern>
            </providers>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="console"/>
        <appender-ref ref="logstash"/>
    </root>
```

对 Logstash 的支持主要通过名为 `logstash` 的 Appender 实现，内容并不复杂，主要是对日志信息的格式化处理，上面为了方便调试和查看，我们先将 JSON 格式的日志输出到文件中。

完成上面的改造之后，我们再将快速入门的示例运行起来，并发起对 `trace-1` 的接口访问。此时可以在 `trace-1` 和 `trace-2` 的工程目录下发现有一个 `build` 目录，下面分别创建了以各自应用名称命名的 JSON 文件，该文件就是在 `logback-spring.xml` 中配置的名为 `logstash` 的 Appender 输出的日志文件，其中记录了类似下面格式的 JSON

日志：


```
{"@timestamp":"2016-12-04T06:57:58.970+00:00","severity":"INFO","service":"trace-1","trace":"589ee5f7b860132f","span":"a9e891273affb7fc","exportable":"false","pid":"19756","thread":"http-nio-9101-exec-1","class":"c.d.TraceApplication$$EnhancerBySpringCGLIB$$a9604da6","rest":"===<call trace-1>==="}
{"@timestamp":"2016-12-04T06:57:59.061+00:00","severity":"INFO","service":"trace-1","trace":"589ee5f7b860132f","span":"2df8511ddf3d79a2","exportable":"false","pid":"19756","thread":"http-nio-9101-exec-1","class":"o.s.c.a.AnnotationConfigApplicationContext","rest":"Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@64951f38: startup date [Sun Dec 04 14:57:59 CST 2016]; parent: org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@4b8c8f15"}
```


除了可以通过上面的方式生成 JSON 文件外，还可以使用 LogstashTcpSocketAppender 将日志内容直接通过 Tcp Socket 输出到 Logstash 服务端，比如：

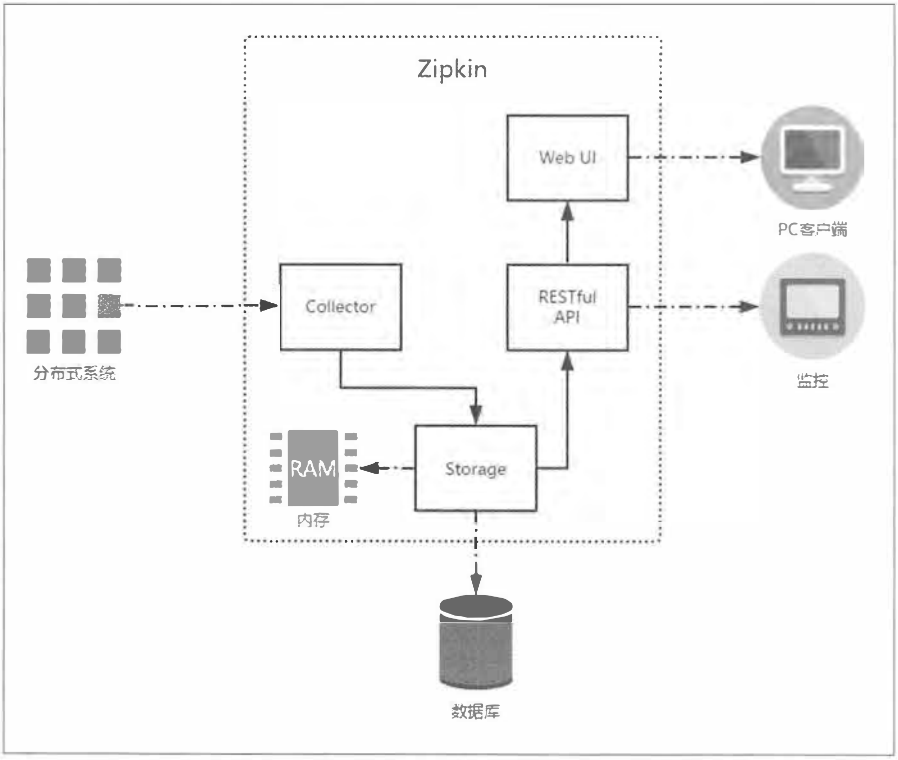
```
<appender name="logstash" class="net.logstash.logback.appender.LogstashTcpSocketAppender">
    <destination>127.0.0.1:9250</destination>
    ...
</appender>
```

与 Zipkin 整合

虽然通过 ELK 平台提供的收集、存储、搜索等强大功能，我们对跟踪信息的管理和使用已经变得非常便利。但是，在 ELK 平台中的数据分析维度缺少对请求链路中各阶段时间延迟的关注，很多时候我们追溯请求链路的一个原因是为了找出整个调用链路中出现延迟过高的瓶颈源，或为了实现对分布式系统做延迟监控等与时间消耗相关的需求，这时候类似 ELK 这样的日志分析系统就显得有些乏力了。对于这样的问题，我们就可以引入 Zipkin 来得以轻松解决。

Zipkin 是 Twitter 的一个开源项目，它基于 Google Dapper 实现。我们可以使用它来收集各个服务器上请求链路的跟踪数据，并通过它提供的 REST API 接口来辅助查询跟踪数据以实现对分布式系统的监控程序，从而及时发现系统中出现的延迟升高问题并找出系统性能瓶颈的根源。除了面向开发的 API 接口之外，它还提供了方便的 UI 组件来帮助我们直观地搜索跟踪信息和分析请求链路明细，比如可以查询某段时间内各用户请求的处理时间等。

下图展示了 Zipkin 的基础架构，它主要由 4 个核心组件构成。



- Collector：收集器组件，它主要处理从外部系统发送过来的跟踪信息，将这些信息转换为 Zipkin 内部处理的 Span 格式，以支持后续的存储、分析、展示等功能。
- Storage：存储组件，它主要处理收集器接收到的跟踪信息，默认会将这些信息存储在内存中。我们也可以修改此存储策略，通过使用其他存储组件将跟踪信息存储到数据库中。
- RESTful API：API 组件，它主要用来提供外部访问接口。比如给客户端展示跟踪信息，或是外接系统访问以实现监控等。
- Web UI：UI 组件，基于 API 组件实现的上层应用。通过 UI 组件，用户可以方便而又直观地查询和分析跟踪信息。

HTTP 收集

在 Spring Cloud Sleuth 中对 Zipkin 的整合进行了自动化配置的封装，所以我们可以很轻松地引入和使用它。下面我们来详细介绍一下 Sleuth 与 Zipkin 的基础整合过程，主要分为以下两步。

第一步：搭建 Zipkin Server

- 创建一个基础的 Spring Boot 应用，命名为 zipkin-server，并在 pom.xml 中引入 Zipkin Server 的相关依赖，具体如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <dependency>
        <groupId>io.zipkin.java</groupId>
        <artifactId>zipkin-server</artifactId>
    </dependency>
    <dependency>
        <groupId>io.zipkin.java</groupId>
        <artifactId>zipkin-autoconfigure-ui</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- 创建应用主类 ZipkinApplication，使用@EnableZipkinServer 注解来启动 Zipkin Server，具体如下：

```
@EnableZipkinServer
@SpringBootApplication
public class ZipkinApplication {

    public static void main(String[] args) {
        SpringApplication.run(ZipkinApplication.class, args);
    }

}
```

- 在 application.properties 中做一些简单配置，比如设置服务端口号为 9411（客户端整合时，自动化配置会连接 9411 端口，所以在服务端设置了端口为 9411 的话，客户端可以省去这个配置）。

```
spring.application.name=zipkin-server
server.port=9411
```

创建完上述工程之后，我们将其启动起来，并访问 http://localhost:9411/，可以看到如下图所示的 Zipkin 管理页面：

第二步：为应用引入和配置 Zipkin 服务

在完成了 Zipkin Server 的搭建之后，我们还需要对应用做一些配置，以实现将跟踪信息输出到 Zipkin Server。我们以之前实现的 trace-1 和 trace-2 为例，对它们做以下改造。

- 在 trace-1 和 trace-2 的 pom.xml 中引入 spring-cloud-sleuth-zipkin 依赖，具体如下所示。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

- 在 trace-1 和 trace-2 的 application.properties 中增加 Zipkin Server 的配置信息，具体如下所示（如果在 zip-server 应用中，我们将其端口设置为 9411，并且均在本地调试的话，也可以不配置该参数，因为默认值就是 http://localhost:9411）。

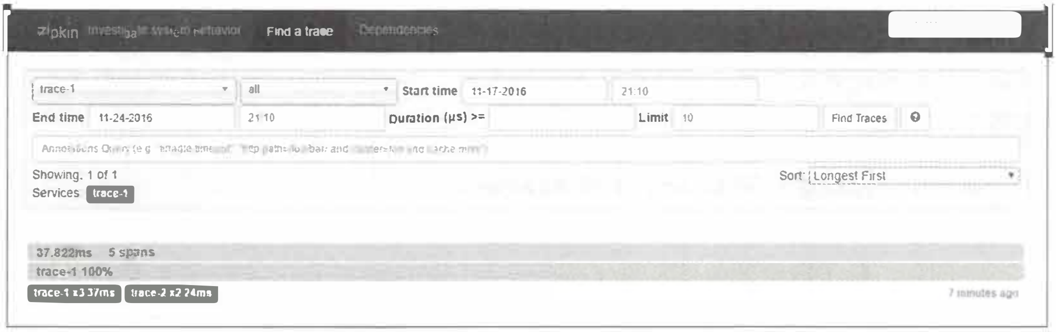
```
spring.zipkin.base-url=http://localhost:9411
```

测试与分析

到这里我们已经完成了接入 Zipkin Server 的所有基本工作，可以继续将 eureka-server、trace-1 和 trace-2 启动起来，然后做一些测试，以对它的运行机

制有一些初步的理解。

我们先来向 `trace-1` 的接口发送几个请求 `http://localhost:9101/trace-1`。当在日志中出现跟踪信息的最后一个值为 `true` 的时候，说明该跟踪信息会输出给 Zipkin Server，所以此时可以在 Zipkin Server 的管理页面中选择合适的查询条件，单击 `Find Traces` 按钮，就可以查询出刚才在日志中出现的跟踪信息了（也可以根据日志中的 Trace ID，在页面右上角的输入框中来搜索），页面如下所示。



单击下方 `trace-1` 端点的跟踪信息，还可以得到 Sleuth 跟踪到的详细信息，其中包括我们关注的请求时间消耗等。

单击导航栏中的 `Dependencies` 菜单，还可以查看 Zipkin Server 根据跟踪信息分析生成的系统请求链路依赖关系图，如下图所示。

消息中间件收集

Spring Cloud Sleuth 在整合 Zipkin 时，不仅实现了以 HTTP 的方式收集跟踪信息，还实现了通过消息中间件来对跟踪信息进行异步收集的封装。通过结合 Spring Cloud Stream，我们可以非常轻松地让应用客户端将跟踪信息输出到消息中间件上，同时 Zipkin 服务端从消息中间件上异步地消费这些跟踪信息。

接下来，我们基于之前实现的 trace-1 和 trace-2 应用以及 zipkin-server 服务端做一些改造，以实现通过消息中间件来收集跟踪信息。改造的内容非常简单，只需要对项目依赖和配置文件做一些调整马上就能实现。下面我们分别对客户端和服务端的改造内容做详细说明。

第一步：修改客户端 trace-1 和 trace-2

- 为了让 trace-1 和 trace-2 在产生跟踪信息之后，能够将抽样记录输出到消息中间件，除了需要之前引入的 spring-cloud-starter-sleuth 依赖之外，还需要引入 Zipkin 对 Spring Cloud Stream 的扩展依赖 spring-cloud-sleuth-stream 以及基于 Spring Cloud Stream 实现的消息中间件绑定器依赖。以使用 RabbitMQ 为例，我们可以加入如下依赖：

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-sleuth-stream</artifactId>
</dependency>

<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

- 在 application.properties 配置中去掉 HTTP 方式实现时使用的 spring.zipkin.base-url 参数，并根据实际部署情况，增加消息中间件的相关配置，比如下面这些关于 RabbitMQ 的配置信息：

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=springcloud
spring.rabbitmq.password=123456
```

第二步：修改 zipkin-server 服务端

为了让 zipkin-server 服务端能够从消息中间件中获取跟踪信息，我们只需要在 pom.xml 中引入针对消息中间件收集封装的服务端依赖 spring-cloud-sleuth-zipkin-

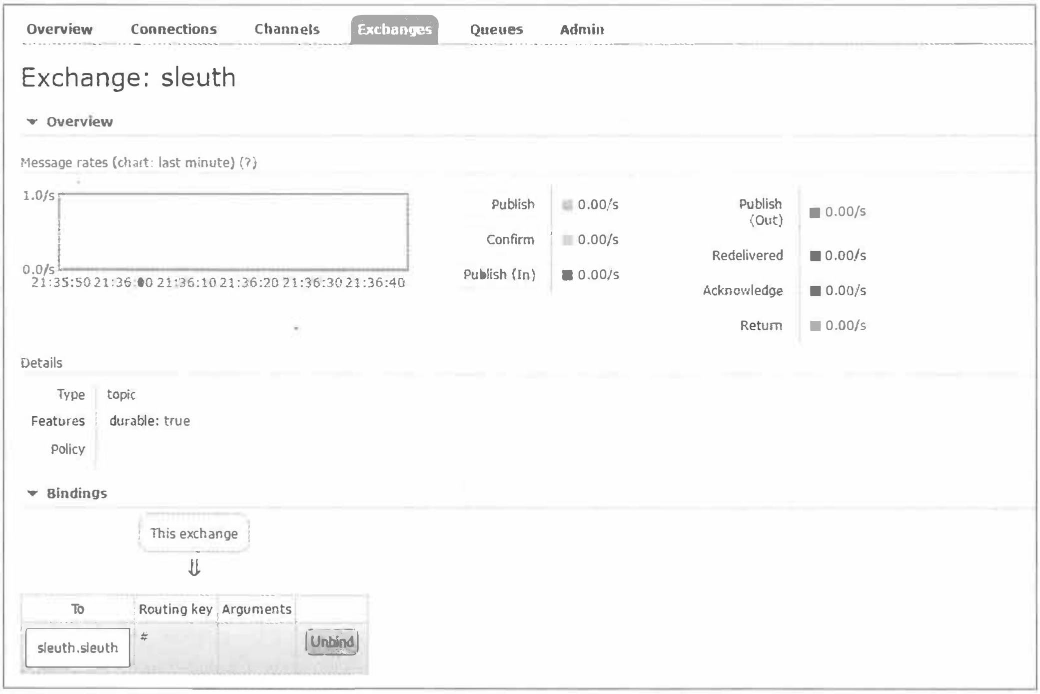
stream，同时为了支持具体使用的消息中间件，我们还需要引入针对消息中间件的绑定器实现。比如以使用 RabbitMQ 为例，我们可以在依赖中增加如下内容：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>

<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
```

其中，spring-cloud-sleuth-zipkin-stream 依赖是实现从消息中间件收集跟踪信息的核心封装，其中包含了用于整合消息中间件的核心依赖、Zipkin 服务端的核心依赖，以及一些其他通常会被使用的依赖（比如，用于扩展数据存储的依赖、用于支持测试的依赖等）。但是，需要注意的是，这个包里并没有引入 zipkin 的前端依赖 zipkin-autoconfigure-ui，为了方便使用，我们在这里也引用了它。

测试与分析

在完成了上述改造内容之后，我们继续将 eureka-server、trace-1 和 trace-2、zipkin-server 都启动起来，同时确保 RabbitMQ 也处于运行状态。此时，我们可以在 RabbitMQ 的控制页面中看到一个名为 sleuth 的交换器，它就是 Zipkin 的消息中间件收集器实现使用的默认主题。



最后，我们使用之前的验证方法，通过向 trace-1 的接口发送几个请求 http://localhost:9101/trace-1。当有被抽样收集的跟踪信息时（调试时我们可以设置 AlwaysSampler 抽样机制来让每个跟踪信息都被收集），我们可以在 RabbitMQ 的控制页面中发现有消息被发送到了 sleuth 交换器中。同时再到 Zipkin 服务端的 Web 页面中也能够搜索到相应的跟踪信息，那么我们使用消息中间件来收集跟踪信息的任务到这里就完成了。

收集原理

在本节内容之前，我们已经对如何引入 Sleuth 跟踪信息和搭建 Zipkin 服务端分析跟踪延迟的过程做了详细的介绍，相信大家对于 Sleuth 和 Zipkin 已经有了一定的感性认识。接下来，我们介绍一下关于 Zipkin 收集跟踪信息的过程细节，以帮助我们更好地理解 Sleuth 生产跟踪信息以及输出跟踪信息的整体过程和工作原理。

数据模型

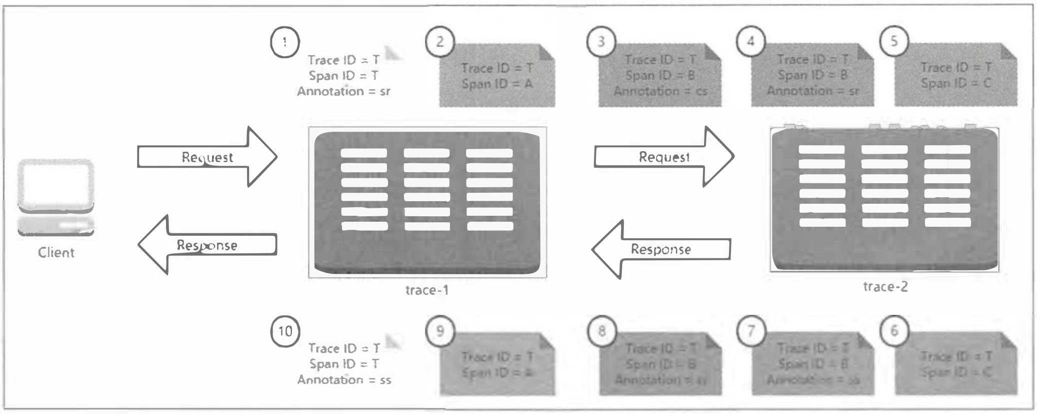
我们先来看看 Zipkin 中关于跟踪信息的一些基础概念。由于 Zipkin 的实现借鉴了 Google 的 Dapper，所以它们有着类似的核心术语，主要有下面几项内容。

- Span：它代表了一个基础的工作单元。我们以 HTTP 请求为例，一次完整的请求过程在客户端和服务端都会产生多个不同的事件状态（比如下面所说的 4 个核心 Annotation 所标识的不同阶段）。对于同一个请求来说，它们属于一个工作单元，所以同一 HTTP 请求过程中的 4 个 Annotation 同属于一个 Span。每一个不同的工作单元都通过一个 64 位的 ID 来唯一标识，称为 Span ID。另外，在工作单元中还存储了一个用来串联其他工作单元的 ID，它也通过一个 64 位的 ID 来唯一标识，称为 Trace ID。在同一条请求链路中的不同工作单元都会有不同的 Span ID，但是它们的 Trace ID 是相同的，所以通过 Trace ID 可以将一次请求中依赖的所有依赖请求串联起来形成请求链路。除了这两个核心的 ID 之外，Span 中还存储了一些其他信息，比如，描述信息、事件时间戳、Annotation 的键值对属性、上一级工作单元的 Span ID 等。
- Trace：它是由一系列具有相同 Trace ID 的 Span 串联形成的一个树状结构。在复杂的分布式系统中，每一个外部请求通常都会产生一个复杂的树状结构的 Trace。
- Annotation：它用来及时地记录一个事件的存在。我们可以把 Annotation 理解为一个包含有时间戳的事件标签，对于一个 HTTP 请求来说，在 Sleuth 中定义了下面 4 个核心 Annotation 来标识一个请求的开始和结束。
  - cs（Client Send）：该 Annotation 用来记录客户端发起了一个请求，同时它也标识了这个 HTTP 请求的开始。
  - sr（Server Received）：该 Annotation 用来记录服务端接收到了请求，并准备开始处理它。通过计算 sr 与 cs 两个 Annotation 的时间戳之差，我们可以得到当前 HTTP 请求的网络延迟。
  - ss（Server Send）：该 Annotation 用来记录服务端处理完请求后准备发送请求响应信息。通过计算 ss 与 sr 两个 Annotation 的时间戳之差，我们可以得到当前服务端处理请求的时间消耗。
  - cr（Client Received）：该 Annotation 用来记录客户端接收到服务端的回复，同时它也标识了这个 HTTP 请求的结束。通过计算 cr 与 cs 两个 Annotation 的时间戳之差，我们可以得到该 HTTP 请求从客户端发起到接收服务端响应的总时间消耗。
- BinaryAnnotation：它用来对跟踪信息添加一些额外的补充说明，一般以键值对的方式出现。比如，在记录 HTTP 请求接收后执行具体业务逻辑时，此时并没有默认的 Annotation 来标识该事件状态，但是有 BinaryAnnotation 信息对其进行补充。

收集机制

在理解了 Zipkin 的各个基本概念之后，下面我们结合前面章节中实现的例子来详细介绍和理解 Spring Cloud Sleuth 是如何对请求调用链路完成跟踪信息的生产、输出和后续处理的。

首先，我们来看看 Sleuth 在请求调用时是怎样来记录和生成跟踪信息的。下图展示了本节中实现示例的运行全过程：客户端发送了一个 HTTP 请求到 `trace-1`，`trace-1` 依赖于 `trace-2` 的服务，所以 `trace-1` 再发送一个 HTTP 请求到 `trace-2`，待 `trace-2` 返回响应之后，`trace-1` 再组织响应结果返回给客户端。



在上图的请求过程中，我们为整个调用过程标记了 10 个标签，它们分别代表了该请求链路运行过程中记录的几个重要事件状态。根据事件发生的时间顺序我们为这些标签做了从小到大的编号，1 代表请求的开始、10 代表请求的结束。每个标签中记录了一些上面提到过的核心元素：Trace ID、Span ID 以及 Annotation。由于这些标签都源自一个请求，所以它们的 Trace ID 相同，而标签 1 和标签 10 是起始和结束节点，它们的 Trace ID 与 Span ID 是相同的。

根据 Span ID，我们可以发现在这些标签中一共产生了 4 个不同 ID 的 Span，这 4 个 Span 分别代表了这样 4 个工作单元。

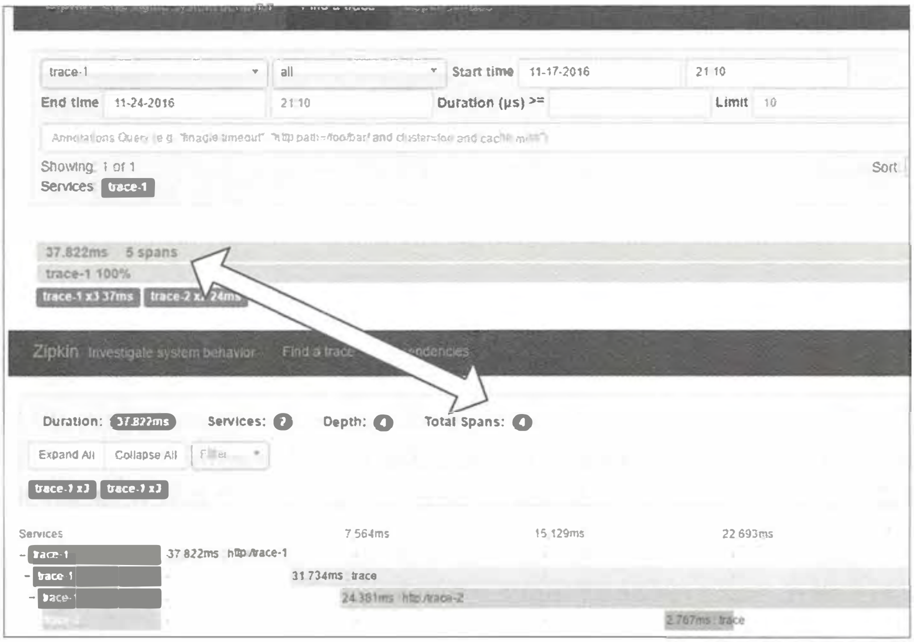
- Span T：记录了客户端请求到达 `trace-1` 和 `trace-1` 发送请求响应的两个事件，它可以计算出客户端请求响应过程的总延迟时间。
- Span A：记录了 `trace-1` 应用在接收到客户端请求之后调用处理方法的开始和结束两个事件，它可以计算出 `trace-1` 应用用于处理客户端请求时，内部逻辑花费的时间延迟。
- Span B：记录了 `trace-1` 应用发送请求给 `trace-2` 应用、`trace-2` 应用接收请求，`trace-2` 应用发送响应、`trace-1` 应用接收响应 4 个事件，它可以计算出 `trace-1` 调用 `trace-2` 的总体依赖时间（cr - cs），也可以计算出 `trace-1` 到

trace-2 的网络延迟（sr - cs），还可以计算出 trace-2 应用用于处理客户端请求的内部逻辑花费的时间延迟（ss - sr）。

- Span C：记录了 trace-2 应用在接收到 trace-1 的请求之后调用处理方法的开始和结束两个事件，它可以计算出 trace-2 应用处理来自 trace-1 的请求时，内部逻辑花费的时间延迟。

在图中展现的这个 4 个 Span 正好对应了 Zi pkin 查看跟踪详细页面中的显示内容，它们的对应关系如下图所示。

仔细的读者可能还有这样一个疑惑：我们在 Zi pkin 服务端查询跟踪信息时（如下图所示），在查询结果页面中显示的 spans 是 5，而单击进入跟踪明细页面时，显示的 Total Spans 又是 4，为什么会出现 span 数量不一致的情况呢？



实际上这两边的 span 数量内容有不同的含义，查询结果页面中的 5 spans 代表了总共接收的 Span 数量，而在详细页面中的 Total Spans 则是对接收 Span 进行合并后的结果，也就是前文中我们介绍的 4 个不同 ID 的 Span 内容。下面我们来详细研究一下 Zipkin 服务端收集客户端跟踪信息的过程，看看它到底收到了哪些具体的 Span 内容，从而来理解 Zipkin 中收集到的 Span 总数量。

为了更直观地观察 Zipkin 服务端的收集过程，我们可以对之前实现的消息中间件方式收集跟踪信息的程序进行调试。通过在 Zipkin 服务端的消息通道监听程序中增加断点，我们能清楚地知道客户端都发送了什么信息到 Zipkin 的服务端。在 spring-cloud-sleuth-zipkin-stream 依赖包中的代码并不多，很容易就能找到定义消息通道监听的实现类：org.springframework.cloud.sleuth.zipkin.stream.ZipkinMessageListener。它的具体实现如下所示，其中 SleuthSink.INPUT 定义了监听的输入通道，默认会使用名为 sleuth 的主题，我们也可以通过 Spring Cloud Stream 的配置对其进行修改。

```
@MessageEndpoint
@Conditional(NotSleuthStreamClient.class)
public class ZipkinMessageListener {

    final Collector collector;

    @ServiceActivator(inputChannel = SleuthSink.INPUT)
    public void sink(Spans input) {
        List<zipkin.Span> converted = ConvertToZipkinSpanList.convert(input);
        this.collector.accept(converted, Callback.NOOP);
    }

    ...

}
```

从通道监听方法的定义中我们可以看到，Sleuth 与 Zipkin 在整合的时候是由两个不同的 Span 定义的，一个是消息通道的输入对象 org.springframework.cloud.sleuth.stream.Spans，它是 Sleuth 中定义的用于消息通道传输的 Span 对象。每个消息中包含的 Span 信息定义在 org.springframework.cloud.sleuth.Span 对象中，但是真正在 Zipkin 服务端使用的并非这个 Span 对象，而是 Zipkin 自己的 zipkin.Span 对象。所以，在消息通道监听处理方法中，对 Sleuth 的 Span 做了处理，每次接收到 Sleuth 的 Span 之后就将其转换成 Zipkin 的 Span。

下面我们尝试在 sink(Spans input) 方法实现的第一行代码中加入断点，并向 trace-1 发送一个请求，触发跟踪信息发送到 RabbitMQ 上。此时我们通过 DEBUG 模式

可以发现消息通道中都接收到了两次输入，一次来自 trace-1，一次来自 trace-2。下面两张图分别展示了来自 trace-1 和 trace-2 输出的跟踪消息，其中 trace-1 的跟踪消息包含了 3 条 Span 信息，trace-2 的跟踪消息包含了 2 条 Span 信息，所以在这个请求调用链上，一共发送了 5 个 Span 信息，也就是我们在 Zipkin 搜索结果页面中看到的 Spans 的数量信息。

点开一个具体的 Span 内容，我们可以看到如下所示的结构，它记录了 Sleuth 中定义的 Span 详细信息，包括该 Span 的开始时间、结束时间、Span 的名称、Trace ID、Span ID、Tags（对应 Zipkin 中的 BinaryAnnotation）、Logs（对应 Zipkin 中的 Annotation）等之前提到过的核心跟踪信息。

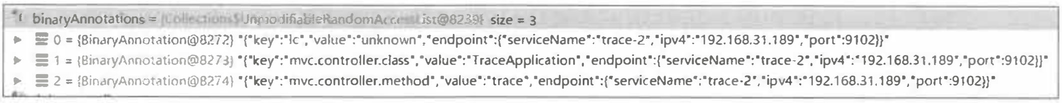
介绍到这里仔细的读者可能会有一个疑惑，在明细信息中展示的 Trace ID 和 Span ID 的值为什么与列表展示的概要信息中的 Trace ID 和 Span ID 的值不一样呢？实际上，Trace ID 和 Span ID 都是使用 long 类型存储的，在 DEBUG 模式下查看其明细时自然是 long 类型，也就是它的原始值，但是在查看 Span 对象的时候，我们看到的是通过 `toString()` 函数处理过的值。从 Sleuth 的 Span 源码中我们可以看到如下定义，在输出 Trace ID 和 Span ID 时都调用了 `idToHex` 函数将 long 类型的值转换成了十六进制的字符串值，所以在 DEBUG 时我们会看到两个不一样的值。

```
public String toString() {
    return "[Trace: " + idToHex(this.traceId) + ", Span: " + idToHex(this.spanId)
        + ", Parent: " + getParentIdIfPresent() + ", exportable:" + this.exportable + "]";
}

public static String idToHex(long id) {
    return Long.toHexString(id);
}
```

在接收到 Sleuth 之后，我们将程序继续执行下去，可以看到经过转换后的 Zipkin 的 Span 内容，它们保存在一个名为 `converted` 的列表中，具体内容如下所示：

上图展示了转换后的 Zipkin Span 对象的详细内容，可以看到很多熟悉的名称，也就是之前介绍的关于 Zipkin 中的各个基本概念。而这些基本概念的值我们也都可以在之前 Sleuth 的原始 Span 中找到，其中 Annotations 和 BinaryAnnotations 有一些特殊。在 Sleuth 定义的 Span 中没有使用相同的名称，而是使用了 logs 和 tags 来命名。从这里的详细信息中，我们可以直观地看到 Annotations 和 BinaryAnnotations 的作用，其中 Annotations 中存储了当前 Span 包含的各种事件状态以及对应事件状态的时间戳，而 BinaryAnnotations 则存储了对事件的补充信息，比如上图中存储的就是该 HTTP 请求的细节描述信息，除此之外，它也可

以存储对调用函数的详细描述（如下图所示）。



下面我们再来详细看看通过调试消息监听程序接收到的这 5 个 Span 内容。首先，可以发现，每个 Span 中都包含有 3 个 ID 信息，其中除了标识 Span 自身的 ID 以及用来标识整条链路的 traceId 之外，还有一个之前没有提过的 parentId，它是用来标识各 Span 父子关系的 ID（它的值来自于上一步执行单元 Span 的 ID）。通过 parentId 的定义我们可以为每个 Span 找到它的前置节点，从而定位每个 Span 在请求调用链中的确切位置。在每条调用链路中都有一个特殊的 Span，它的 parentId 为 null，这类 Span 我们称它为 Root Span，也就是这条请求调用链的根节点。为了弄清楚这些 Span 之间的关系，我们可以从 Root Span 开始来整理出整条链路的 Span 内容。下表展示了我们从 Root Span 开始，根据各个 Span 的父子关系整理出的结果：

Host	Span ID	Parent Span ID	Annotation	BinaryAnnotation
trace-1	e9a933ec50d180d6	null	[sr, ss]	
trace-1	c48122fa096bffe8	e9a933ec50d180d6		[trace-1,TraceApplication,trace]
trace-1	1ae2e9a317faa422	c48122fa096bffe8	[cs, cr]	
trace-2	1ae2e9a317faa422	c48122fa096bffe8	[sr, ss]	
trace-2	36194e4182985c4e	1ae2e9a317faa422		[trace-2,TraceApplication,trace]

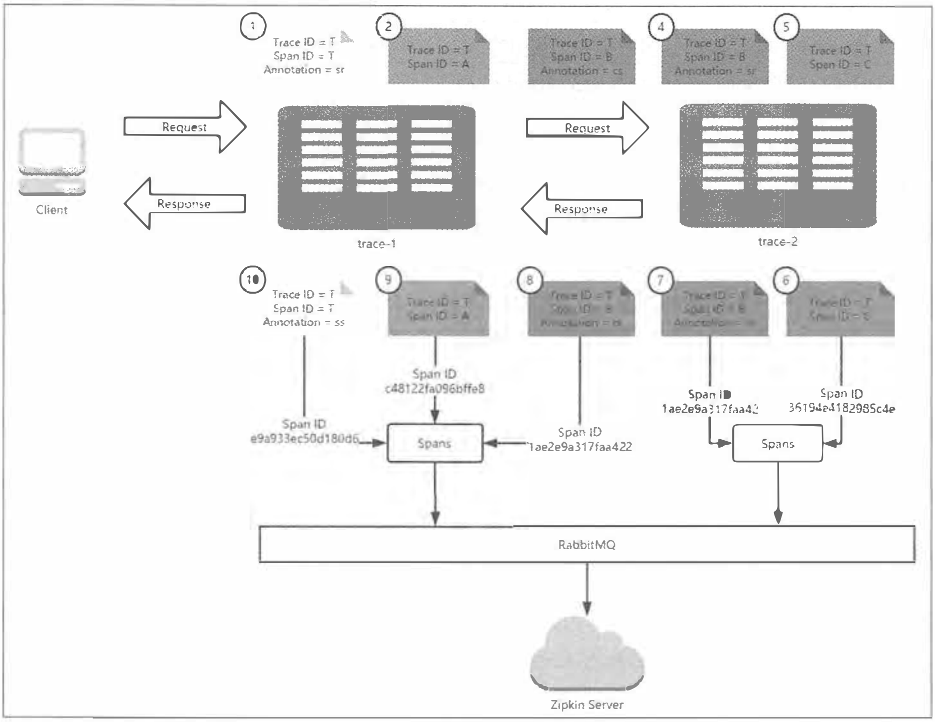
上表中的 Host 代表了该 Span 是从哪个应用发送过来的；Span ID 是当前 Span 的唯一标识；Parent Span ID 代表了上一执行单元的 Span ID；Annotation 代表了该 Span 中记录的事件（这里主要用来记录 HTTP 请求的 4 个阶段，表中内容进行了省略处理，只记录了 Annotation 名称（sr 代表服务端接收请求，ss 代表服务端发送请求，cs 代表客户端发送请求，cr 代表客户端接收请求），省略了一些其他细节信息，比如服务名、时间戳、IP 地址、端口号等信息）；BinaryAnnotation 代表了事件的补充信息（Sleuth 的原始 Span 记录更为详细，Zipkin 的 Span 处理后会去掉一些内容，对于有 Annotation 标识的信息，不再使用 Binary Annotation 补充，在上表中我们只记录了服务名、类名、方法名，同样省略了一些其他信息，比如时间戳、IP 地址、端口号等信息）。

通过收集到的 Zipkin Span 详细信息，我们很容易将它们与本节开始时介绍的一次调用链路中的 10 个标签内容联系起来。

- `Span ID = T` 的标签有 2 个，分别是序号 1 和 10，它们分别表示这次请求的开始和结束。它们对应了上表中 ID 为 `e9a933ec50d180d6` 的 Span，该 Span 的内容在标签 10 执行结束后，由 `trace-1` 将标签 1 和 10 合并成一个 Span 发送给 Zipkin Server。

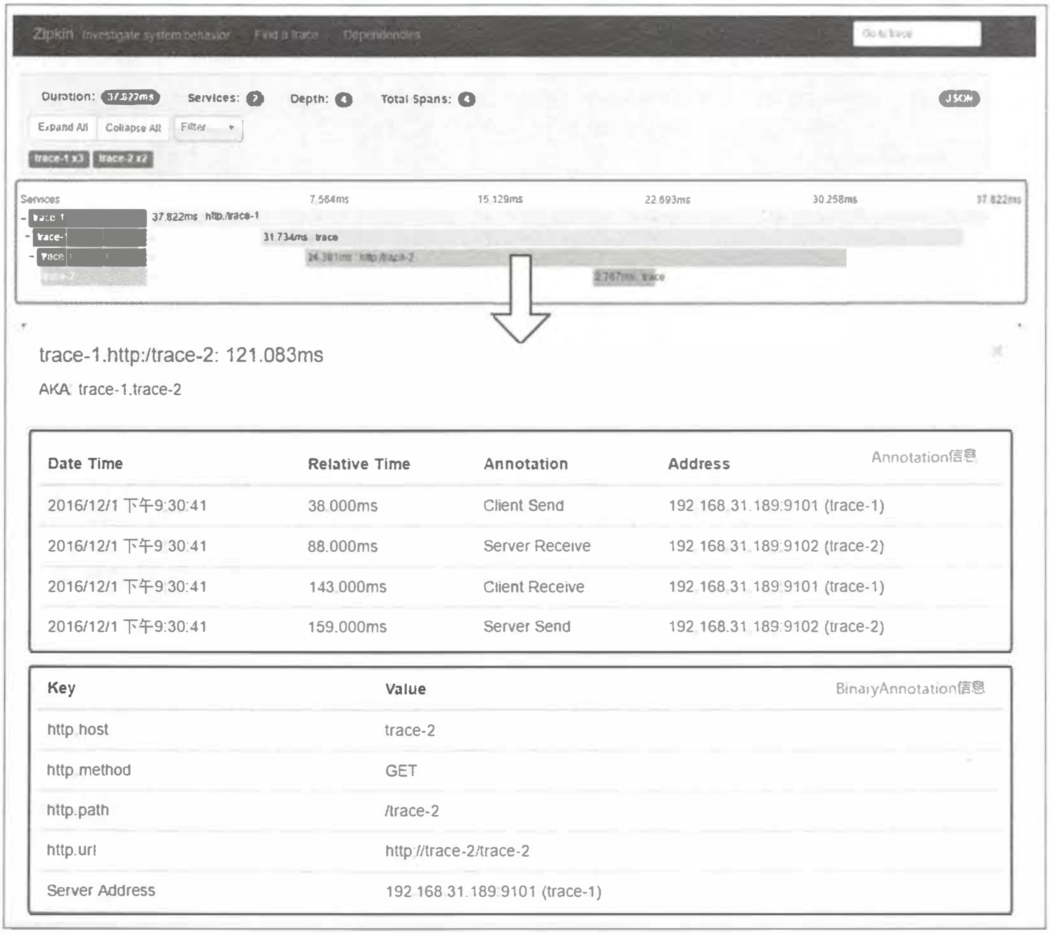
- `Span ID = A` 的标签有 2 个，分别是序号 2 和 9，它们分别表示了 `trace-1` 请求接收后，具体处理方法调用的开始和结束。该 Span 的内容在标签 9 执行结束后，由 `trace-1` 将标签 2 和 9 合并成一个 Span 发送给 Zipkin Server。
- `Span ID = B` 的标签有 4 个，分别是序号 3、4、7、8，该 Span 比较特殊，它的产生跨越了两个实例，其中标签 3 和 8 是由 `trace-1` 生成的，而标签 4 和 7 则是由 `trace-2` 生成的，所以该标签会拆分成两个 Span 内容发送给 Zipkin Server。`trace-1` 会在标签 8 结束的时候将标签 3 和 8 合并成一个 Span 发送给 Zipkin Server，而 `trace-2` 会在标签 7 结束的时候将标签 4 和 7 合并成一个 Span 发送给 Zipkin Server。
- `Span ID = C` 的标签有 2 个，分别是序号 5 和 6，它们分别表示了 `trace-2` 请求接收后，具体处理方法调用的开始和结束。该 Span 的内容在标签 6 执行结束后，由 `trace-2` 将标签 2 和 9 合并成一个 Span 发送给 Zipkin Server。

所以，根据上面的分析，Zipkin 总共会收到 5 个 Span：一个 Span T，一个 Span A，两个 Span B，一个 Span C。结合之前请求链路的标签图和这里的 Span 记录，我们可以总结出如下图所示的 Span 收集过程，读者可以参照此图来理解 Span 收集过程的处理逻辑以及各个 Span 之间的关系。



虽然，Zipkin 服务端接收到了 5 个 Span，但就如前文中分析的那样，其中有两个 Span ID=B 的标签，由于它们来自于同一个 HTTP 请求（trace-1 对 trace-2 的服务调用），概念上它们属于同一个工作单元，因此 Zipkin 服务端在前端展现分析详情时会将这两个 Span 合并显示，而合并后的 Span 数量就是在请求链路详情页面中 Total Spans 的数量。

下图是本章示例的一个请求链路详情页面，在页面中显示了各个 Span 的延迟统计，其中第三条 Span 信息就是 trace-1 对 trace-2 的 HTTP 请求调用，通过单击它可以查看该 Span 的详细信息。单击后会以模态框的方式弹出 Span 详细信息（如图下半部分所示），在弹出框中详细展示了 Span 的 Annotation 和 BinaryAnnotation 信息，在 Annotation 区域我们可以看到它同时包含了 trace-1 和 trace-2 发送的 Span 信息，而在 BinaryAnnotation 区域则展示了该 HTTP 请求的详细信息。



数据存储

默认情况下，Zipkin Server 会将跟踪信息存储在内存中，每次重启 Zipkin Server 都会使之前收集的跟踪信息丢失，并且当有大量跟踪信息时我们的内存存储也会成为瓶颈，所以通常情况下我们都需要将跟踪信息对接到外部存储组件中去，比如使用 MySQL 存储。

Zipkin 的 Storage 组件中默认提供了对 MySQL 的支持，所以我们可以很轻松地为 zipkin-server 增加 MySQL 存储功能。下面我们详细介绍基于消息中间件实现的 zipkin-server 应用，对其进行 MySQL 存储扩展的详细过程。

第一步：为 zipkin-server 添加依赖

为了让 zipkin-server 能够访问 MySQL 数据库，我们需要在它的 pom.xml 文件中增加如下依赖，以支持对 MySQL 的访问：

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
   <groupId>org.jooq</groupId>
   <artifactId>jooq</artifactId>
   <version>3.8.0</version>
</dependency>
```

这里需要注意加入 org.jooq 的依赖，并不是因为缺少这个依赖而引入它，而是为了解决该版本存在的一个 Bug。当不引入该依赖的时候，会使用默认的依赖，但是默认情况下引入的依赖并非 3.8.0，当有跟踪信息被 zipkin-server 收集并入库时会报如下错误：

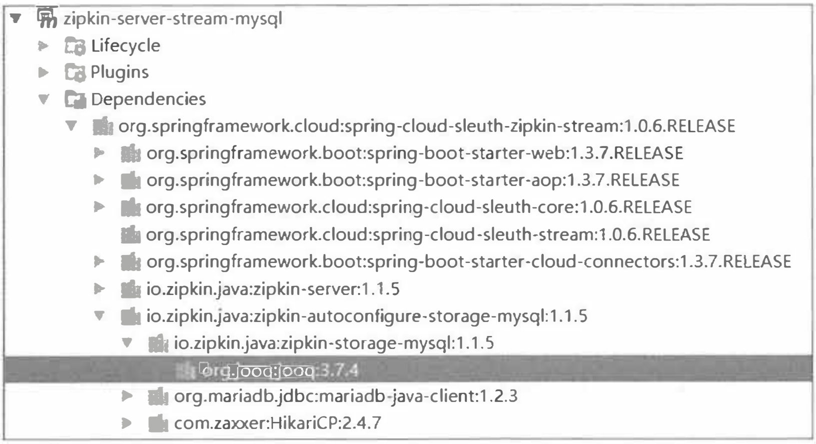
```
    o.s.c.s.z.stream.ZipkinMessageListener    : Cannot store spans
[e7aca79a855ff80a.30671a9177f24801<:114ffd85950b117b,
e7aca79a855ff80a.114ffd85950b117b<:d8ac47672741466c] due to VerifyError(class
zipkin.storage.mysql.internal.generated.tables.ZipkinSpans overrides final method
getSchema.()Lorg/jooq/Schema;)


    java.lang.VerifyError: class
zipkin.storage.mysql.internal.generated.tables.ZipkinSpans overrides final method
getSchema.()Lorg/jooq/Schema;
    ...
    o.s.c.s.i.m.MessagingSpanExtractor      : Deprecated trace headers detected.
Please upgrade Sleuth to 1.1 or start sending headers present in the TraceMessageHeaders
class
    o.s.c.s.z.stream.ZipkinMessageListener    : Cannot store spans
```

```
[8ce053ed8c2410a0.d6b63783a69485d8<:9ebc8131ebdb07fc,
8ce053ed8c2410a0.9ebc8131ebdb07fc<:8ce053ed8c2410a0,
8ce053ed8c2410a0.8ce053ed8c2410a0<:8ce053ed8c2410a0] due to
VerifyError(zipkin/storage/mysql/internal/generated/tables/ZipkinSpans)
```

从报错信息中我们可以看到 `VerifyError(zipkin/storage/mysql/internal/generated/tables/ZipkinSpans)` 这一句，从 `ZipkinSpans` 的源码中我们可以看到如下内容：

```
@Generated(
    value = {
        "http://www.jooq.org",
        "jOOQ version:3.8.0"
    },
    comments = "This class is generated by jOOQ"
)
```

从注解中可以看到，`ZipkinSpans` 对象是通过 jOOQ 3.8.0 生成出来的，但是当工程不加入上面的 `org.jooq` 依赖时，jOOQ 的版本是 3.7.4，正是因为这里的版本不一致导致了上面错误的出现。所以，如果读者在使用其他版本的 Spring Cloud Sleuth 出现类似错误时，可以看看是否是由这个原因引起的。



第二步：在 MySQL 中创建用于 Zipkin 存储的 Schema

这里需要注意一点，Zipkin Server 实现关系型数据库存储时，不同版本对数据库表结构都有一些变化。当跨版本使用时很容易出现各种整合问题，所以尽量使用对应版本的脚本来创建 Schema。同时，Zipkin 实现的 MySQL 存储仅在 MySQL 5.6-5.7 版本中测试过，所以尽量使用对应版本的 MySQL，以避免产生不可预知的问题。

本书中我们使用了 Spring Cloud 的 Brixton.SR5 版本，通过查看依赖关系可以知道它使用的 Zipkin 版本为 1.1.5，所以我们可以从 GitHub 上找到该版本并下载创建表结构的脚本。

另外，也可以在本地依赖中找到该脚本，下图展示了 mysql.sql 脚本在本地依赖中的具体位置：

从上图中我们还可以知道，MySQL 的存储支持是通过 zipkin-autoconfigure-storage-mysql 依赖实现的，但是我们之前为什么没有引入该依赖呢？这是由于我们改造的工程基础是消息中间件实现的示例，之前提到过，在该示例中引入的 spring-cloud-sleuth-zipkin-stream 依赖包含了各个常用的依赖组件，其中就包括了对 MySQL 的支持依赖，所以我们这里并不需要再手工添加它。当然，如果不是基于该示例，而是通过 HTTP 实现的收集示例来改造时，就需要自己引入对 zipkin-autoconfigure-storage-mysql 的依赖了。

在获取创建表结构的 SQL 文件之后，我们可以在 MySQL 中手工创建名为 zipkin 的 Schema，并运行 mysql.sql 脚本来创建表结构。除此之外，也可以通过在程序中进行配置的方式让其自动初始化，只需要在 application.properties 中增加如下配置：

```
spring.datasource.schema=classpath:/mysql.sql
spring.datasource.url=jdbc:mysql://localhost:3306/zipkin
spring.datasource.username=root
spring.datasource.password=123456
spring.datasrouce.continueOnError=true
spring.datasrouce.initialize=true
```

通过启动程序，Spring Boot 的 JDBC 模块会自动地为我们根据指定的 SQL 文件来创建表结构。我们可以得到如下两张表。

- zipkin_spans：存储 Span 信息的表。
- zipkin_annotations：存储 Annotation 信息的表。

第三步：切换存储类型

通过第二步，我们已经让 zipkin-server 连接到 MySQL，并且创建了用于存储跟踪信息的 Schema。下面我们只需要再做一个简单配置，让 zipkin-server 的存储切换到 MySQL 即可，具体配置如下：

```
zipkin.storage.type=mysql
```

测试与验证

到这里，我们就已经完成了将 `zipkin-server` 从内存存储跟踪信息切换为 MySQL 存储跟踪信息的改造。最后，我们继续使用之前的验证方法，通过向 `trace-1` 的接口发送几个请求 `http://localhost:9101/trace-1`，当有被抽样收集的跟踪信息时（调试时可以设置 `AlwaysSampler` 抽样机制来让每个跟踪信息都被收集），查看 MySQL 中的两张表，可以得到类似下面的数据信息。

- `zipkin_spans` 表

trace_id	id	name	parent_id	debug	start_ts	duration
8972329404569714412	-3039763029959719084	trace	7221594919888949682	(NULL)	1480759142774000	5239
8972329404569714412	1210319574519141610	trace	8972329404569714412	(NULL)	1480759142759000	23632
8972329404569714412	7221594919888949682	http:/trace-2	1210319574519141610	(NULL)	1480759142767000	13538
8972329404569714412	8972329404569714412	http:/trace-1	(NULL)	(NULL)	1480759142751000	32404

- `zipkin_annotations` 表

trace_id	span_id	a_key	a_value		a_type	a_timestamp	endpoint_ipv4	endpoint_port	endpoint_service_name
8972329404569714412	7221594919888949682	[illegible]	(NULL)	0K	-1	1480759142761000	-1062731674	9101	trace-1
8972329404569714412	7221594919888949682	cr	(NULL)	0K	-1	1480759142779000	-1062731674	9101	trace-1
8972329404569714412	7221594919888949682	http.host	trace-2	7B	6	1480759142761000	-1062731674	9101	trace-1
8972329404569714412	7221594919888949682	http.method	GET	3B	6	1480759142761000	-1062731674	9101	trace-1
8972329404569714412	7221594919888949682	http.path	/trace-2	8B	6	1480759142761000	-1062731674	9101	trace-1
8972329404569714412	7221594919888949682	http.url	http://trace-2	22B	6	1480759142761000	-1062731674	9101	trace-1
8972329404569714412	7221594919888949682	sa	□	1B	0	1480759142761000	-1062731674	9101	trace-1
8972329404569714412	1210219574519141610	lc	unknown	7B	6	1480759142759000	-1062731674	9101	trace-1
8972329404569714412	1210319574519141610	mvc.controller.class	TraceApplication	16B	6	1480759142759000	-1062731674	9101	trace-1
8972329404569714412	1210319574519141610	mvc.controller.method	trace	5B	6	1480759142759080	-1062731674	9101	trace-1
8972329404568714412	6972329404569714412	sr	(NULL)	0K	-1	1480759142751000	-1062731674	9101	trace-1
8972329404569714412	8972329404569714412	ss	(NULL)	0K	-1	1480759142783000	-1062731674	9101	trace-1
8972329404569714412	-3039763029959719084	lc	unknown	7B	6	1480759142774000	-1062731674	9102	trace-2
8972329404569714412	-3039763029959719084	mvc.controller.class	TraceApplication	16B	6	1480759142774000	-1062731674	9102	trace-2
8972329404569714412	-3039763029959719084	mvc.controller.method	trace	5B	6	1480759142774000	-1062731674	9102	trace-2
8972329404569714412	7221594919888949682	sr	(NULL)	0K	-1	1480759142767000	-1062731674	9102	trace-2
8972329404569714412	7221594919888949682	ss	(NULL)	0K	-1	1480759142780000	-1062731674	9102	trace-2

表中所存储的信息我们已经非常熟悉，之前分析的内容都可以在这两张表中体现出来。比如 Span 的数量问题，从 `zipkin_spans` 表中，我们可以看到一次请求调用链路的跟踪信息产生了 4 条 span 数据，也就是说，在入 `zipkin_spans` 表的时候，已经对收集的 span 信息进行了合并，所以在查询详细信息时，不需要每次都来计算合并 span。而在 `zipkin_annotations` 表中，通过 `span_id` 字段可以关联到每个具体工作单元的详细信息，同时根据 `endpoint_service_name` 和 `span_id` 字段还可以计算出一次请求调用链路中总共接收到的 span 数量。

Zipkin 在存储方面除了对 MySQL 有扩展组件之外，还实现了对 Cassandra 和 ElasticSearch 的支持扩展。具体的整合方式与 MySQL 的整合类似，读者可自行查阅 Zipkin 的官方文档做进一步的了解。

API 接口

Zipkin 不仅提供了 UI 模块让用户可以使用 Web 页面来方便地查看跟踪信息，它还提供了丰富的 RESTful API 接口供用户在第三方系统中调用来定制自己的跟踪信息展示或监控。我们可以在 Zipkin Server 启动时的控制台或日志中找到 Zipkin 服务端提供的 RESTful API 定义，比如下面的日志片段：

```
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/dependencies],methods=[GET],produces=[application/json]}" ...
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/trace/{traceId}],methods=[GET],produces=[application/json]}" ...
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/traces],methods=[GET],produces=[application/json]}" ...
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/services],methods=[GET]}" ...
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/spans],methods=[GET]}" ...
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/spans],methods=[POST]}" ...
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/spans],methods=[POST],consumes=[application/x-thrift]}" ...
```

可以看到 Zipkin Server 提供的 API 接口都以`/api/v1`路径作为前缀，它们的具体功能整理如下：

接口路径	请求方式	接口描述
`/dependencies`	GET	用来获取通过收集到的 Span 分析出的依赖关系
`/services`	GET	用来获取服务列表
`/spans`	GET	根据服务名来获取所有的 Span 名
`/spans`	POST	向 Zipkin Server 上传 Span
`/trace/{traceId}`	GET	根据 Trace ID 获取指定跟踪信息的 Span 列表
`/traces/`	GET	根据指定条件查询并返回符合条件的 trace 清单

更多关于接口的请求参数和请求返回格式等细节说明，可以通过访问 Zipkin 官方的 API 页面 `http://zipkin.io/zipkin-api/`来查看，帮助我们根据自身系统架构来访问 Zipkin Server 以定制自己的 Dashboard 或监控系统。实际上，Zipkin 的 UI 模块也是基于 RESTful API 接口来实现的，有兴趣的读者可以通过浏览器的开发者模式来查看每个页面发起的请求，以此作为调用样例来参考。

附录 A

Starter POMs

下面整理了针对 Spring Boot 1.3.7 版本的 Starter POMs 模块。

模块名	功能描述
spring-boot-starter	Spring Boot 核心模块，包括自动配置支持、日志、YAML
spring-boot-starter-actuator	监控模块，可以帮助我们监控和管理应用
spring-boot-starter-amqp	通过 spring-rabbit 支持 Advanced Message Queuing Protocol
spring-boot-starter-aop	对 AOP 面向切面编程的支持，包括 spring-aop 和 AspectJ
spring-boot-starter-artemis	通过 Apache Artemis 来支持 Java Message Service API
spring-boot-starter-batch	对 Spring Batch 的支持，包含 HSQLDB 数据库
spring-boot-starter-cache	对 Spring Cache 的抽象支持
spring-boot-starter-cloud-connectors	对 Spring Cloud Connectors 的支持，可以通过它简单地连接到云平台的服务，比如 Cloud Foundry 和 Heroku
spring-boot-starter-data-elasticsearch	对 ElasticSearch 搜索和分析引擎的支持，包括 spring-data-elasticsearch
spring-boot-starter-data-gemfire	对 GemFire 分布式存储的支持，包含 spring-data-gemfire
spring-boot-starter-data-jpa	对 Java Persistence API 的支持，包括 spring-data-jpa、spring-orm 和 Hibernate
spring-boot-starter-data-mongodb	对 MongoDB NoSQL 数据库的支持，包括 spring-data-mongodb
spring-boot-starter-data-rest	通过 spring-data-rest-webmvc 支持以 REST 形式暴露 Spring Data 的 repositories
spring-boot-starter-data-solr	对 Apache Solr 搜索平台的支持，包括 spring-data-solr
spring-boot-starter-freemarker	对 FreeMarker 模板引擎的支持
spring-boot-starter-groovy-templates	对 Groovy 模板引擎的支持
spring-boot-starter-hateoas	通过 spring-hateoas 支持基于 HATEOAS 的 RESTful 服务
spring-boot-starter-hornetq	通过 HornetQ 支持 Java Message Service API

续表

模块名	功能描述
spring-boot-starter-integration	对 spring-integration 模块的支持
spring-boot-starter-jdbc	对 JDBC 数据库连接的支持
spring-boot-starter-jersey	支持 Jersey 的 RESTful Web Services 框架
spring-boot-starter-jta-atomikos	通过 Atomikos 实现 JTA 分布式事务
spring-boot-starter-jta-bitronix	通过 Bitronix 实现 JTA 分布式事务
spring-boot-starter-mail	对 javax.mail 的支持
spring-boot-starter-mobile	对 spring-mobile 的支持
spring-boot-starter-mustache	对 Mustache 模板引擎的支持
spring-boot-starter-redis	对 Redis 数据库的支持，包含 spring-redis
spring-boot-starter-security	对 spring-security 的支持
spring-boot-starter-social-facebook	对 spring-social-facebook 的支持
spring-boot-starter-social-linkedin	对 spring-social-linkedin 的支持
spring-boot-starter-social-twitter	对 spring-social-twitter 的支持
spring-boot-starter-test	支持常用的测试框架依赖模块，包括 JUnit、Hamcrest 和 Mockito
spring-boot-starter-thymeleaf	对 thymeleaf 模板引擎的支持，包含与 Spring 的整合
spring-boot-starter-velocity	对 Velocity 模板引擎的支持
spring-boot-starter-web	对全栈 Web 开发的支持，包含 Tomcat 和 spring-webmvc
spring-boot-starter-websocket	对 WebSocket 开发的支持
spring-boot-starter-ws	对 Spring Web Service 的支持

后记

2017 年 1 月 8 日，总算赶在过年前基本完成了本书的编写。自博客上发表第一篇关于 Spring Cloud 的文章开始，持续地研究、实践与输出相关内容已经有半年之多。而编写本书是从 2016 年 9 月开始的，本以为采用当时最新的 Release 版本可以让本书内容紧跟潮流，然而 Spring Cloud 的发展远比我所想象的要快得多。在这 4 个月的时间中，Brixton 版本从 SR5 升级到了 SR7，而 Camden 版本也开始进入 Release 阶段，并持续升级到了 SR3。惊叹 Spring Cloud 高速发展的同时，也让我坚定了对 Spring Cloud 的支持。相信在未来的几年内，在企业应用架构的选择上，Spring Cloud 一定会开始崭露头角。

虽然本书内容是基于 Brixton SR5 版本实现的，但经调试均可以兼容目前最新的 Brixton SR7 和 Camden SR3 版本。不过，对于部分写法在 Camden 中被标注了废弃状态，比如 Spring Cloud Stream 中的一些注解等。对于 Spring Cloud 的更新内容，相信也是大部分读者希望可以持续获得的信息，由于笔者自身也在实践这部分内容，因此在完成本书之后，我也会持续关注 Spring Cloud 的发展，并继续开始更新博客内容，主要关注于 Spring Cloud 的新特性以及实践中碰到的一些问题。读者可以关注我的博客：http://blog.didispace.com，或是我的微信公众号：didispace，以获得最新的分享信息。

在去年创建的 Spring Cloud 中文社区论坛（http://bbs.springcloud.com.cn），目前在许进的牵头和帮助下，已经与一些其他优秀资源做了整合，目前归入 Spring Cloud 中国社区（http://springcloud.cn）统一管理。由于 2016 年大家都忙于各自的工作，还没有对社区的运作（包括文档贡献和知识分享等）做出一定的规划。在 2017 年，我们会抽出时间来规划一些对 Spring Cloud 知识分享、文档贡献等相关的公益性活动，以帮助所有 Spring Cloud 爱好者与实践者共同成长。

下面是关于本书的代码案例资源，主要维护于 GitHub 和开源中国上。如果读者在阅读过程中对文章内容或示例有任何疑问均可以在这里向我提问。

- GitHub：https://github.com/dyc87112/SpringCloudBook
- 开源中国：http://git.oschina.net/didispace/SpringCloudBook

在本书的最后，我要特别感谢我的家人在这段时间内给予的支持，让我能够挤出更多的时间来完成本书的编写。同时，也要感谢 Spring Cloud 中国社区的朋友、关注我博客的读者，以及交流群中给我提出过不少优秀建议的网友，正是因为有了大家的支持，才能让我一直坚持写完本书。即便如此，本书还是无法覆盖 Spring Cloud 的全部，但是后续我会通过博客等其他渠道紧跟 Spring Cloud 的发展，分享更多实战经验和研究成果。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

Spring Cloud
微服务实战

这本书让我对 Spring Cloud 各组件的使用提升了一个层次，也让我对其实现原理有了初步认识，建议所有打算将 Spring Cloud 用于生产的朋友好好读一下。

——王天青 DaoCloud 首席架构师

为数不多全面讲解 Spring Cloud 微服务的中文书，囊括实施微服务需要深入了解的各个轮子，是一本需要仔细研读的精品之作。

——朱清 冰鉴科技信息技术部总监

书如其人，的确是一本好书，特别是基于技术实践的阐述中又不失对“微服务化”理论层面的讲解以及发展演进的说明，我会推荐给更多的人。

——李俊涛 上海驻云科技执行总监

本书从微服务架构概念出发，结合 Spring Cloud 的解决方案，深入浅出地剖析了其在构建微服务架构中所需的各个基础设施和技术要点，适合正在或打算实施微服务的团队。

——南志文 百联全渠道研发总监

Spring Cloud 是没有足够资金投入或者技术储备的技术团队的福音，其一站式解决方案可从容应对业务的快速发展。这本实战书籍给我们带来了无限惊喜。

——覃罗春 德比软件产品开发负责人

Spring Boot/Cloud 生态的出现为行业提供了一站式解决方案。而要想真正解决架构选型和维护的难题，每一位开发者和架构师都应重视本书中介绍的案例和源码分析。

——程超 合众支付资深技术专家

作为 Spring Cloud 中国社区联合发起人和首批 Spring Cloud 实践与布道者，本书作者积累了大量生产实战经验，并将其通过书中案例奉献给实施微服务架构的企业和开发者。

——许进（xujin.org） Spring Cloud 中国社区创始人/中间件高级研发工程师

微服务概念深入人心，而“全家桶” Spring Cloud 几乎可实现所有微服务功能。这本用于快速上手的工作手册，犹如一颗种子，生根发芽后必将长成一棵参天大树。

——吴峻申 上海青客机器人有限公司架构师

上架建议：Java微服务

ISBN 978-7-121-31301-1

定价：89.00元

策划编辑：张春雨
责任编辑：刘 舫
封面设计：李 玲